NAG1-613

February 1992

UILU-ENG-92-2202
CRHC-92-01

Center for Reliable and High-Performance Computing

IN-61-CR
72157
p. 133

ISE -- AN INTEGRATED SEARCH ENVIRONMENT -- THE MANUAL

Lon-Chan Chu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
UILU-ENG-92-2202 CRHC-92-01

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Coordinated Science Lab
University of Illinois

6b. OFFICE SYMBOL *(If applicable)*
N/A

7a. NAME OF MONITORING ORGANIZATION
NASA
NSF

6c. ADDRESS *(City, State, and ZIP Code)*
1101 W. Springfield Avenue
Urbana, IL 61801

7b. ADDRESS *(City, State, and ZIP Code)*
Washington DC 12200
Washington DC 12200

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
7a

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
7b

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
ISE--an integrated search environment --the manual

12. PERSONAL AUTHOR(S)
Chu, Lon-Chan

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
1992 February 5

15. PAGE COUNT

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
integraed search environment, hierarchical search, problem-independent, user-friendly

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

In this manual, we describe the software package ISE (acronym for *Integrated Search Environment*), a tool that implement hierarchical searches with meta-control. ISE actually is a collection of problem-independent routines to support solving searches. Mainly, these routines are core routines for solving a search problem and they handle the control of searches and maintain the statistics related to searches. By separating the problem-dependent and problem-independent components in ISE, new search methods based on a combination of existing methods can be developed easily by coding a single master control program. Further, new applications solved by searches can be developed by coding the problem-dependent parts and reusing the problem-independent parts already developed. Potential users of ISE would be designers of new application solvers and new search algorithms, and users of experimenting application solvers and search algorithms. The ISE is designed to be user-friendly and information rich. In this manual, the organization of ISE is described and several experiments carried out on ISE are also described.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE *(Include Area Code)*

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ISE -- AN INTEGRATED SEARCH ENVIRONMENT
THE MANUAL

Lon-Chan Chu
Center for Reliable and High-performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, Illinois 61801
chu@aquinas.csl.uiuc.edu

ABSTRACT

In this manual, we describe the software package ISE (acronym for *Integrated Search Environment*), a tool that implements hierarchical searches with meta-control. ISE actually is a collection of problem-independent routines to support search processes. Mainly, these routines are core routines for solving a search problem and they handle the control of searches and maintain the statistics related to searches. By separating the problem-dependent and problem-independent parts in ISE, new search methods can be implemented by calling existing methods and they can be developed easily by coding the meta-control. Further, new applications can be developed by only coding the problem-dependent parts. Potential users of ISE would be designers of new application solvers and new search algorithms, and users of experimenting them. ISE is designed to be user-friendly and information rich. In this manual, the organization of ISE is described and some sample runs are also shown.

Research partially supported by National Aeronautics and Space Administration under contract NCC 2-481 and by National Science Foundation under grant MIP 88-10584.

Technical Report CRHC-92-1.

1. INTRODUCTION

This manual describes the usage of our tool, ISE (*Integrated Search Environment*), that facilitates the coding of solving search problems. ISE is the implementation of the hierarchical search processes proposed by Wah [10]. ISE actually is a collection of problem-independent routines to support search solving. Mainly, these routines are *core* routines of solving a search problem, since they handle the control of searches and maintain the statistics related to searches. Potential users of ISE would be designers of new application solvers, designers of new search algorithm, and persons that use search algorithms and application solvers.

New application solvers only need to implement the problem-dependent part and need to interface some parameters related to problem-related characteristics, like the data type of cost values and the problem-dependent part of search nodes, such that the core routines can do things right. Once a new application solver is implemented in ISE, then this application problem can be solved by any of search strategies and search algorithms implemented in ISE in the past or in the future.

New search algorithm designers also can implement their new search algorithms by manipulating builtin search primitives provided by ISE. A search primitive is the most basic search method such as best-first search and depth-first search.

Application users will find the friendliness of ISE and can reconfigurate the search processes via the command line, like search strategy, search algorithm, and profiling status.

In this report, the *traveling salesperson problem* (TSP) and IDA* search algorithm [4] will be used as running examples to illustrate the procedure of coding new application problem solvers and new search algorithms, respectively, on ISE.

This manual is organized as follows. Section 2 briefly describes the architecture of ISE. Section 3 describes WISE core routines. Section 4 describes the development of application problem solvers. Section 5 describes the interface between ISE and new application solvers. Section 6 describes the procedure of implementing new search algorithms on ISE. Section 7 describes sample runs and sample profiles. Finally, section 8 draws the conclusion.

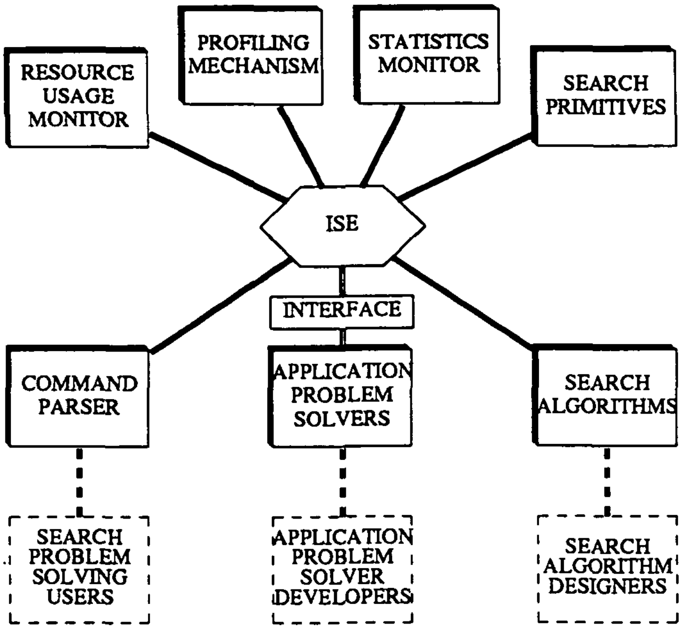

Figure 2.1 ISE Architecture.

2. ISE ARCHITECTURE

The ISE architecture can be illustrated in Figure 2.1. Potential users can add new search algorithms and new application solvers into ISE with minimal interfacing and minimal efforts in coding. Further, application users can solve the search problem by using any search algorithm in ISE simply via the parameters in the command line. In this sense, ISE is an open software and it can be expanded by adding new application solvers and new search algorithms.

Physically, ISE is a collection of routines about search primitives, search control, statistics, and profiling mechanism. They are stored over several directories: `include/`, `primitive/`, `algorithm/`, `kernel/`, `open/`, `interface/`, and `solver/`.

Directory `include/` contains all necessary macros, constants, data structures, and declarations of global variables. Directory `primitive/` contains all the search primitives which is the building block of all search algorithms in ISE. The basic primitives currently implemented are `bfs_primitive`: best-first search primitive.

`dfs_primitive`: depth-first search primitive.

`gdfs_primitive`: guided depth-first search primitive.

`gbb_primitive`: generic branch-and-bound search primitive.

`band_primitive`: band search primitive.

Directory `alogirhtm/` contains all the routines of search algorithms. Default search algorithms means pure BFS, DFS, GDFS, and B&B. In fact, BFS and GDFS can be represented by B&B with best-first and depth-first selection functions, respectively. However, the memory behavior of these searches are entirely different; therefore, they are intended to be seperated. Other search algorithms are implemented by manipulating search primitives or algorithms.

Directory `kernel/` contains all *kernel* routines about search maintenances, resource usage monitoring, resource constraint checking, and statistics monitoring.

Directory `open/` contains routines which dynamically allocate chunks of space for search nodes. The deallocations are done by pooling the deallocated space for future allocations. Also, this directory contains the manager of active lists in the search. Note that the active lists may be different for different search primitives due to efficiency reasons.

Directory `interface/` contains all the routines which parse context-free, user-friendly command lines and generate particular statistics specified by users.

All the application solvers are stored in directory `solver/`. The problem-dependent routines for one specific problem are kept in one directory, for example `solver/ats.pgm/` for the TSP, which contains several small files including routines of search node allocation, search node evaluation, and search node decomposition.

The *search primitives* are the basic search units. Currently, best-first, depth-first, guided depth-first, generic branch-and-bound search, and band search primitives are provided in ISE. From now on, let "bfs", "dfs", "gdfs", "gbb", and "bs" denote best-first, depth-first, guided depth-first, generic branch-and-bound search, and band search, respectively.

Before a search primitive can be called, some information called *search parameters* must be ready, and they include

(1) *Resource Constraints.* Constraints of time, space, and cumulative space-time (CST) cost can be specified such that the search will be terminated when any of these hard constraints is violated.

(2) *Degree of Approximation.* The expected degree of approximation of the solution can be used as a parameter to the search primitives. The degree of approximation is in the range of $[0, \infty)$. The solution is called the optimal solution if the degree of approximation is zero; otherwise, the solution is called α-approximately semi-optimal solution, if the degree of approximation is α.

(3) *Statistics Status.* The search process can be measured *logically* or *physically.* The measurement includes time, space, and CST cost. The physical time is measured by the timer provided by the operating system. The physical space is measured by the number of words in storage used for storing search nodes. The physical CST is the cumulative product of physical space and physical time. The logical time is measured by the number of search nodes expanded or the number of search nodes generated according to user's specification. The logical space is measured by the number of active search nodes. The logical CST is the cumulative product of logical space and logical time.

(4) *Profiling Status.* The search process can be profiled at run time. The profiling status gives the information about what kind of profiling needs to be done. The profiling covers time, space, CST cost, and degree of approximation.

(5) *Algorithm-Specific Constraints.* The algorithm-specific constraints differ from the hard resource constraints in the way that when the algorithm-specific constraints are violated only the search primitive is terminated, while the entire search algorithm may be terminated if any of the hard resource constraints is violated. In practice, the algorithm-specific constraint may be the expected optimal value, *e.g.* threshold in IDA* [3].

A search algorithm can be defined recursively by either a sequence of search primitives or search algorithms. A sequence is a tuple of one or more ordered items. A search algorithm is said *in ISE format* if it is represented by a sequence of search primitives or search algorithms. For example, IDA* [3] can be represented by a sequence of ''dfs'' primitives with proper setting of resource constraints and algorithm-specific constraints.

By feeding proper parameters, we can do profiling *intra* search primitives or *inter* search primitives. We can implement new search algorithms by simply manipulating search primitives. One powerful support in ISE is that any search algorithm of hierarchical fashion can be easily implemented by manipulating builtin search algorithms and search primitives. Several of the famous search algorithms are represented here in ISE format as examples of manipulating search primitives and search algorithms.

A* = <bfs>
DFS = <dfs>
IDA = <dfs-1, dfs-2, ..., dfs-n>
sTCA* = <bfs-1, bfs-2, ..., bfs-n>
pTCA* = <bfs> ; with profiling intra search primitive.
dTCA* = <bfs> ; with profiling intra search primitive.
sTCGD* = <gdfs-1, gdfs-2, ..., gdfs-n>
pTCGD* = <sTCGD*, gdfs> ; with profiling inter search primitives.
Beam Search = <gbb> ; by properly defining selection and pruning.
Hill-Climbing Search = <gbb> ; by properly defining selection and pruning.

3. ISE CORE ROUTINES

In this section, the problem independent part of ISE is described. The core routines are kept in several function-specific directories. Roughly, these core routines can be classified into several categories: definitions and declarations (`include/`), search primitives (`primitive/`), search algorithms (`algorithm/`), ISE kernel (`kernel/`), memory management (`open/`), IO transducer (`interface/`). These categories are described following.

3.1. Definitions and Declarations

The definitions of data structures are put in `include/define.h` such that all routines can access these definitions. The problem-independent part of the search node is also defined in this file; while the problem-dependent part is defined by users. The Makefile will automatically include the definition of the problem-dependent part and make the definition of node structure complete. The declarations of all global variables and the routines of initializing data structures are put in `include/var.c`. The corresponding external declarations are in include/var.h. The limiting constants, *e.g.* maximum integer, maximum long, and maximum floating-point number, are defined in include/limits.h, which is used to patch the systems that do not support complete limiting constants, *e.g.* Sun 3/60 workstation. Some global variables used for debugging are defined in include/debug.h and declared in include/debug.c.

There is a trick for all definition and declaration routines such that all of them will be compiled once. The trick is setting a compiler directive to check reentrance.

```
#ifndef __define_h_
#define __define_h_
<body of define.h>
#endif  __define_h_
```

3.2. Search Primitives

The search primitives supported in ISE include "bfs", "dfs", "gdfs", "gbb", and "bs" primitives. They are located in primitve/bfs.c, primitive/dfs.c, primitive/gdfs.c, primitive/bb.c, and primitive/band.c, respectively. The input arguments to these search primitives are a pointer to a search structure, a pointer to its parent's search structure, and a message consisting of search controls and profiling status.

The search structure, see search_ { ... }, includes resource constraints, cost data type, measurement system, statistics status, algorithm-specific constraints, degree of approximation, and some algorithm-specific information.

The resource constraints are used to limit the resource usage of the search in a way that the search is terminated when any of the resource constraints is violated. If the search is at the top level, *i.e. root search*, then the entire search process is terminated when any of the resource constraints is violated. However, if the search is at the intermediate or leaf level of the hierarchy of searches, *i.e.* at the root, then only the particular search is terminated when any of its resource constraints is violated.

The cost data type indicate the data type of the solution value, *e.g.* integer or float. The measurement system indicates that time, space, and CST cost are either logical or physical. In fact, both physical and logical measures are taken in ISE.

Statistics status keeps track of statistics of search performance like number of search node generated, number of search node expanded, number of search nodes pruned, and number of feasible-solution search nodes. These statistics are very important, since they can be used to determine the performance of a particular search algorithm. The parent's search structure is used only for the statistics tracing.

The algorithm-specific constraints provide another way to terminate the search prematurely. Typically, the expected optimal solution value is one of the soft constraints, which is used in IDA* [3], *i.e.* the threshold.

The degree of approximation is used to solve the problem approximately by pruning more nodes that might lead to the optimal solution. The approximation means the solution is complete but is not optimal, instead of incomplete or partial solution.

All the search primitives have an algorithm-specific entry inside their routines, which allows the search algorithm designers to do some accounting, maintenance, and search process adapting.

The search message has a control flag indicating this particular search primitive should be started freshly or simply resumed. This allows to resume an old search terminated prematurely.

The algorithm-specific entry, the capability of resumption, and the algorithm-specific constraints allow the search algorithm designers to have the freedom to manipulate different styles of search algorithms.

3.3. Search Algorithms

The search algorithm designers can put the source codes of their algorithms to the directory `algorithm/`. Currently existing algorithms include pure BFS, DFS, GDFS, general B&B [4], and band search [2], as well as TCA* (sTCA*, pTCA*, and dTCA*) [7] and TCGD (sTCGD and pTCGD) [8], iterative deepening A* (IDA*) [3], iterative refining A* (IRA*) [5], DFS* [6], real-time search (RTS) [1], and Lawler and Wood's time-constrained approximation search [4].

All these search algorithms can be defined as a sequence of either search primitives or search algorithms. For example, IDA* is a sequence of ''dfs'' primitives, and pTCGD is a sTCGD algorithm (with profiling inter search primitives) followed by a ''gdfs'' primitive. Through the support of search primitives and profiling mechanisms, the search algorithm designers will find that it is very easy to implement their search algorithms by manipulating the search primitives and profiling mechanisms.

3.4. ISE Kernel

The kernel includes the main routine and all the routines about tracing statistics, profiling, and supporting utilities. The statistics include the performance of the search process and the measures of the resource usages. The profiling can be either intra search primitive or inter search primitives. If the profiling is intra search primitive, then the search message fed to the search primitive must have a signal indicating that. If the profiling is inter search primitives, then the profiling routines can be called between two contiguous search primitives. The supporting utilities include a wide spectrum of small routines which make coding easy and data structures abstract.

3.5. Memory Management

The memory management includes the allocation and deallocation of search nodes as well as the handling of the active lists of search nodes.

The allocation and deallocation of search nodes should match the computer architecture and hierarchical memory organization such that the search process will not suffer from frequent swapping or thrashing due to virtual memory faults. In ISE, the search nodes are allocated chunks by chunks, where the chunk size `AllocationSize` is defined by the application solver, because the size of a search node may vary over several orders of magnitude for different applications. In this way, the search nodes have high probability that they are located nearby such that they will not spread over the entire virtual memory. Further, the deallocated search nodes are kept in a linked list for future allocations rather than released to the operating system. In this way, the spatial locality and temporal locality during the search can be reserved.

The active list is a conceptual term, because it is physically different for different search primitives. For the ''bfs'' primitive, the active list should be a B+ tree, which allows logarithmic time to insert or delete a search node in the B+ tree as well as provides easy locating the search nodes whose lower bounds no less than a certain value such that pruning in the active list can be done more easily. However, the general ''bfs'' primitive should allow a set of keys instead of a single key, therefore, the ''bfs'' primitive should use a linked list for its active list due to difficulty in pruning by bounding. If the compiler option `BPLUS_TREE` is defined, then the B+ tree is used; otherwise, the linked list is used.

For both ''gdfs'' and ''gbb'' primitives, the active list is simply a linked list. The reason is that the number of search nodes in the active list of a ''gdfs'' primitive is bounded by the product of the maximum branching degree and the maximum depth of the search tree. The reason of using a linked list for the ''gbb'' primitive is that the active list is ordered by a certain selection function defined by the application solver and the pruning function is also defined by the application solver. Both functions are application problem dependent so that the B+ tree will make pruning very complex. Note that pruning is not necessarily to be bounding, and it can be dominance and approximation. Therefore, the simple linked list is used for the ''gbb'' primitive.

For the ''dfs'' primitive, the active list is simply a stack whose size is bounded by the depth of the search tree. The ''dfs'' primitive is actually a backtracking search and all search nodes are partially expanded instead of being fully expanded as in other search primitives; therefore, a stack must be used for backtracking instead of linked list or B+ tree.

3.6. IO Transducer

The IO transducer consists of the command-line processor and the versatile output generator.

3.6.1. Command Line Processor

The command-line processor in `interface/command.c` parses the command line and transforms the command parameters into the internal format of search controls. Application users can define which search strategies and algorithms are to be used for solving the application. Further, users also can define how many sample problems are to be solved as well as the profiling status and the resource constraints via the command line. The typical command line is following.

```
search :prob iter size seed
       [:virtual | :real]
       [:report [report-file]]
       [:search bfs   [default | alg] |
        :search gdfs  [default | alg] |
        :search dfs   [default | alg] |
        :search band  [default | alg]]
       [:constr time [space [cst]]]
       [:approx approx]
       [:dbg dbg]
       [:io stat [summary [graph-ready [rt-pf [st-pf]]]]]
       [:pf rt [st]]
       [:param num_params p-1 p-2 ... p-n]
```

where '[]' means the parameters are optional and Parameter `:prob` must be specified; otherwise, ISE cannot know what the problem size is, how many sample problems to solve, and how to generate the sample problems.

The measurement of resources is either `:virtual` or `:real`. The former indicates logical measurement and the latter does physical measurement. The `:report` option provides an entry for monitoring the algorithm-specific performance, eg. in IDA* the performance of each iteration can be output to the file specified in `:report`. The `:search bfs default` defines that the search primitive is `bfs` and the search algorithm is `default`, that means that the search algorithm only contains a single search primitive. The `:prob iter size seed` defines that the number of sample problems is `iter`, the sample problem size is `size`, and the seed to the random number generator is `seed`. The `:constr time space cst` defines that the time constraint is `time`, the space constraint is `space`, and the cumulative space-time product constraint is `cst`. The `:approx` defines the degree of approximation. The `:dbg` defines the debugging mode which allows multiple-level details of debugging information. The `:io` defines the names of the output files, where the first one is for the performance statistics in the list format, the second one is for the summary of performance statistics with more detailed information, the third one is for the graph-ready format of performance statistics, the fourth one is for the run-time profile, and the fifth one is for the space-vs-time profile. The `:pf rt st` defines that the run-time profiling status is `rt` and the space-time profiling status is `st`. If the profiling is desired, then the status is `YES`; otherwise, it is `NO`. The `:param` is for algorithm-specific parameters.

If some of these parameters are not specified in the command line, ISE have default values for them. The default command line is as follows.

```
search :virtual
       :report report
       :search bfs default
       :prob undefined undefined undefined
       :constr unlimited unlimited unlimited
       :approx 0.0
       :dbg 0
       :io stat summary graph-ready rt-pf st-pf
       :pf 0 0
```

3.6.2. Output Generator

The output file generator is in `interface/output.c` which can produce the summary of performance of search processes as well as the performance data in the format which can be used directly by the graph package `grap`. The performance items to be put in the graph-ready format can be specified by the application solver in the file named `output.h` in the application-specific directory.

The performance statistics in the list format are generated by the routines in `interface/result.c` which summarize the performance statistics into a list which can be easily used by a LISP program. To be more specific, the items in this list is equal to the number of sample problems solved plus 1. Each item is a list of performance statistics of the search processes solving a sample problem. The first item is the averages of the performance statistics over all sample problems. The others are the performance statistics for all sample problems, one for each.

4. APPLICATION DEVELOPMENT

In this section, the development of a new application program will be described. The user must provide several problem-dependent routines. These problem-dependent routines include:

(1) *Problem Definition*: The problem-dependent data structures as well as their

initialization routines are defined and the corresponding global variables are declared.
(2) *Bounding Function Routines*: They evaluate upper and lower bounding functions, if applicable.
(3) *Sample Problem Generator*: A routine can generate new sample problems upon the problem size and a random seed given.
(4) *Search Node Management Routines*: A collection of routines can (i) allocate search nodes by calling ISE search node allocation facility, (ii) initialize the problem-dependent part of a search node, and (iii) set up the links inside a search node to the problem-dependent size-dependent region.
(5) *Problem-Dependent Search Components*: Feasibility and infeasibility tests as well as search environment initializations are needed.

Currently, several applications have been implemented in ISE, including symmetric traveling salesperson problem (sTSP), knapsack problem (KS), production planning problem (PP), vertex cover problem (VC), weighted completion time problem (WCT), general resource constrained scheduling problem (GRCS), asymmetric traveling salesperson problem (aTSP), maze problem (Maze), and *N*-puzzle problem (Puzzle). The maze generator in ISE is modified from the X11R4 maze generator [3].

5. INTERFACING

The interface between ISE and application solvers are described in this section. The interface includes definitions of data structures, necessary routines, and specifications of parameters.

5.1. Definitions of Data Structures

Several data structures must be defined by the application problem solver developer, since they are problem-dependent. These include the data type of solution values, the allocation chunk size, the problem-dependent part of search nodes, and the solution structure. Take the TSP as an example. The TSP solver must define `domain`, `AllocationSize`, `PDSI_PART` macro, and `solution_` structure, as we can see in Appendix.

5.2. Necessary Routines

Several problem-dependent routines are required to make ISE work correctly. They include sample problem generation, bounding function routines, feasibility and infeasibility tests, adaptive search node decomposition, root generation, search node allocation, search node initialization, search node linkage, solution buffer management, and solution interpretation.

6. DEVELOPMENT OF NEW ALGORITHM

In this section, the procedure of developing a new search algorithm is described. The procedure is simple and it include the following steps.
(1) Map the new search algorithm into a sequence of search primitives.
(2) Set up the bookkeeping and interfacing between contiguous two search primitives.
(3) Translate the sequence of search primitives and the associated bookkeeping and interfacing into the C language forms ISE supports.

Take IDA* as an example. First, IDA* can be represented as a sequence of ''dfs'' primitives. Second, design the algorithm which maintains the threshold setting between contiguous ''dfs'' primitives. Third, translate them into the C language forms ISE supports. The IDA* search algorithm in the ISE format can be found in Appendix.

7. SAMPLE RUNS

In this section, we will show some sample commands and their results.

Assume ISE is compiled for the symmetric TSP by command ''make all''. Assume we wish to solve a fifteen-city TSP instance using best-first search. Then, the command-line input is

```
ts :prob 1 15 1 :search bfs default
```

The summary result in file `summary` is

```
counting system=virtual
time unlimited
```

```
space unlimited
cst unlimited
real time=677
real max space=27280
real cst=1.18474e+07
virtual time=415
virtual max space=155
virtual cst=41221
root approx=0.300167
run-time approx=0
optimal solution value=318.554
cmd-line (or adapted) approx=0
achieved approx=0
run-time approx=0
```

The above summary is quite self-explanatory. The row-wise result in file `graph-ready` is

```
999999999  415  155  41221  0.300167  0  0  318.554  318.554  1e+20
```

The first item is the time constraint, where "999999999" denotes infinity. The second, third, and fourth items are completion time, maximum space used, cumulative space-time product, respectively. The fifth one is the run-time approximation degree of the root node. The sixth and seventh items are the final run-time approximation degree and the final achieved approximation degree, respectively. Both are zeros because the optimal solution is solved. The eighth and ninth items are the incumbent and the global lower bound, respectively. Both are equal to the optimal solution. The last one is the threshold, where "1e+20" denote the infinite threshold.

If we want to use guided depth-first search for solving the same TSP instance. The command is

```
ts :prob 1 15 1 :search gdfs default
```

The summary file becomes

```
counting system=virtual
time unlimited
```

```
space unlimited
cst unlimited
real time=677
real max space=5280
real cst=1.67218e+06
virtual time=421
virtual max space=30
virtual cst=6095
root approx=0.300167
run-time approx=0
optimal solution value=318.554
cmd-line (or adapted) approx=0
achieved approx=0
run-time approx=0
```

The ''graph-ready'' file is cumulative and now becomes

```
999999999  415  155  41221  0.300167  0  0  318.554  318.554  1e+20
999999999  421  30  6095  0.300167  0  0  318.554  318.554  1e+20
```

If we want to use the band search of bandwidth 5 for solving the same TSP instance. The command is

```
ts :prob 1 15 1 :search band default :param 2 5 F
```

where parameter ''F'' means the bandwidth is fixed all the time. The summary file becomes

```
counting system=virtual
time unlimited
space unlimited
cst unlimited
real time=663
real max space=17072
real cst=4.56086e+06
virtual time=415
virtual max space=97
virtual cst=16821
root approx=0.300167
run-time approx=0
optimal solution value=318.554
cmd-line (or adapted) approx=0
```

```
achieved approx=0
run-time approx=0
```

The "graph-ready" file is cumulative and now becomes

```
999999999  415  155  41221  0.300167  0  0  318.554  318.554  1e+20
999999999  421  30  6095  0.300167  0  0  318.554  318.554  1e+20
999999999  415  97  16821  0.300167  0  0  318.554  318.554  1e+20
```

Note that the third one is band search of bandwidth 5. In many cases, band search (cf. data in the third row) can almost use the same amount of time as best-first search (cf. data in the first row), but in general requires less memory. In fact, band search use bounded amount of memory [2].

8. CONCLUDING REMARKS

ISE is still an evolving software to support the research on design of resource-constrained search algorithms. ISE is aimed to support a wide spectrum of search algorithms and application solvers. Its goal is to move the problem-independent part of application solvers into an integrated kernel such that the integrated kernel is transparent to the designers of either new applications or new search algorithms such that they can be done in less time as possible.

ACKNOWLEDGEMENT

I would like to give my great thanks to Professor Benjamin .W. Wah for his constant and helpful discussions on improving ISE model and structures. I also would like to thank Arthur Ieumwananonthachai for his helpful discussions on coding. This work is partially supported by National Aeronautics and Space Administration under contract NCC 2-481 and by National Science Foundation under grant MIP 88-10584.

REFERENCES

[1] L. C. Chu and B. W. Wah, "Optimization in Real Time," *Proc. Real Time Systems Symposium*, IEEE, Nov. 1991.

[2] L.-C. Chu and B. W. Wah, ''Band Search,'' *In preparation*, 1992.

[3] R. Hess, D. Lemke, and M. Weiss, ''Maze,'' *X Version 11 Release 4*, July 1991.

[4] R. E. Korf, ''Depth-First Iterative Deepening: An Optimal Admissible Tree Search,'' *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.

[5] E. L. Lawler and D. W. Wood, ''Branch and Bound Methods: A Survey,'' *Operations Research*, vol. 14, pp. 699-719, ORSA, 1966.

[6] G.-J. Li and B. W. Wah, ''Parallel Iterative Refining A*: An Efficient Search Scheme for Solving Combinatorial Optimization Problems,'' *Proc. Int'l Conf. on Parallel Processing*, St. Charles, IL, Aug. 12-16, 1991.

[7] N. Rao Vempaty, V. Kumar, and R. E. Korf, ''Depth-First vs Best-First Search,'' *Proc. National Conf. on Artificial Intelligence*, AAAI, Anaheim, CA, July 1991.

[8] B. W. Wah and L.-C. Chu, ''TCA*--A Time-Constrained Approximate A* Search Algorithm,'' *Proc. Int'l Workshop on Tools for Artificial Intelligence*, pp. 314-320, IEEE, Nov. 1990.

[9] B. W. Wah and L.-C. Chu, ''TCGD: A Time-Constrained Approximate Guided Depth-First Search Algorithm,'' *Proc. Int'l Computer Symposium*, pp. 507-516, Taiwan, China, Dec. 1990.

[10] B. W. Wah and L. C. Chu, ''Hierarchical Branch and Bound Algorithms: Modeling and Implementations,'' *Int'l J. of Artificial Intelligence Tools*, vol. 1, no. 2, World Scientific Publishers, (accepted to appear) April 1992.

APPENDIX

The source code of ISE software is listed as below. The software consists of ''include'', ''primitive'', ''algorithm'', ''kernel'', ''open'', ''interface'', and ''solver'' directories. The applications include TSP, KS, PP, VC, WCT, GRCS, Maze, and Puzzle. Further, relevant makefiles are also listed.

ISE -- Integrated Search Environment

Any part of this software can be used, modified, or copied only if either Professor Benjamin W. Wah or Mr. Lon-Chan Chu is acknowledged.

This software does NOT provide ANY WARRANTY and even not any implied warranty of merchantability or fitness for a particular purpose.

This software is NOT responsible for any damage of the machine due to execution of this software.

```
Thu Jan 30 15:19:05 CST 1992
::::::::::::::
include/limits.h
::::::::::::::

#ifndef __wise_limits_h_
#define __wise_limits_h_

/** Some UNIX versions support the following limiting numbers:
 **    INT_MAX = 2147483647
 **    LONG_MAX = 2147483647
 **    FLT_MAX  = 3.40282346638528860e+38
 **    DBL_MAX  = 1.7976931348623157e+308               **/

#define         HUGE_INT        (9999999)
#define         HUGE_LONG       (999999999)
#define         HUGE_FLOAT      (1.0e20)
#define         HUGE_DOUBLE     (1.0e50)
#define         HUGE_DEPTH      HUGE_INT

#endif __wise_limits_h_

::::::::::::::
include/define.h
::::::::::::::

#ifndef __wise_define_h_
#define __wise_define_h_

#include        <stdio.h>
#include        <ctype.h>
#include        <string.h>
#include        <math.h>
#include        <sys/types.h>
#include        <sys/times.h>


#define RANDOM_RADIX    ((float) (0x7fffffff))


/* const definitions */
#define NIL                         ('\0')
#define DONT_CARE                   (- 100)


/* predicting & profiling */
#define PROFILING_STEP              5
#define REGRESSION_ORDER            1


/* problem-dependent definitions, if they are forgotten to be defined */
#ifndef MaxProblemSize
#define MaxProblemSize              200
#endif

#ifndef AllocationSize
#define AllocationSize              256
#endif

#ifndef MaxNumberOfProblems
#define MaxNumberOfProblems         100
#endif

#ifndef NumberOfBins
#define NumberOfBins                50
#endif

#ifndef MinFeasibleTime
#define MinFeasibleTime             (problem.size)
#endif

#ifndef FirstMaxDepth
#define FirstMaxDepth               256
#endif


#define is_feasible_time(t)         (t >= MinFeasibleTime)


/* algorithm specific definition */

#ifndef MinValidBin
#define MinValidBin                 4
#endif


/* type conversion shorthand */
#define toshort(a)          ((short) (a))
#define toint(a)            ((int) (a))
#define tolong(a)           ((long) (a))
#define tofloat(a)          ((float) (a))
#define todouble(a)         ((double) (a))

#define min(a,b)            ((a > b) ? b : a)
#define max(a,b)            ((a > b) ? a : b)
#define in(a,b,c)           ((a >= b) && (a <= c))

#define streql(a,b)         (! strcmp(a,b))

#define accu2approx(a)      ((a == 0.0) ? huge_float : 1.0 / a - 1.0)
#define approx2accu(e)      ((e == huge_float) ? 0.0 : 1.0 / (e + 1.0))


/* what the search for */
enum _task_ { LEARNING = 0, CSP = 1 };
typedef enum _task_                 task_;


/* search style */
enum _search_style_ { RESUME = 0, FRESH_START = 1 };
typedef enum _search_style_         search_style_;


/* question answer */
enum _yesno_ { NO = 0, YES = 1 };
typedef enum _yesno_                yesno_;


enum _boolean_ { FALSE = 0, TRUE = 1};
typedef enum _boolean_              boolean_;


/** threshold is defined absolutely or relatively **/
typedef enum { ABSOLUTE = 0, RELATIVE = 1 } metric;


/* search ending info */
enum _search_ending_ {
        SEARCH_IS_ABORTED = 0,
        SEARCH_IS_COMPLETED = 1,
        SEARCH_IS_IDLE = 2
};
typedef enum _search_ending_    search_ending_;


/* domain data types */
enum _datatype_ { INT = 1, LONG = 2, FLOAT = 3, DOUBLE = 4};
typedef enum _datatype_         datatype_;


/* search strategies */
enum _strategy_ { GBB, BFS, DFS, GDFS, BAND };
typedef enum _strategy_         strategy_;


/* search algorithms */
enum _algorithm_ {      DEFAULT,
                        LW,
                        sTCA, pTCA, dTCA,
                        sTCGD, pTCGD,
                        uRTS, bRTS, hRTS,
                        uIRD, bIRD,
                        IRA, IDA, DFS_star,
                        PBFS, PDFS, PGDFS, PGBB, PBAND };
typedef enum _algorithm_        algorithm_;


/* definitions for counting system */
enum _count_ { REAL, VIRTUAL };
typedef enum _count_            count_;


/* profiling signals  */
enum _psignal_ { RUN_TIME = -10, SPACE_VS_TIME = -20 };
typedef enum _psignal_          psignal_;


/* which child to be expanded */
enum _child_ { ALL_CHILDREN = -1, NEXT_CHILD, NEXT_N_CHILDREN };
typedef enum _child_            child_;


/* enumerative type */
enum _enum_type_ { TYPE_NULL = 0, TYPE_I = 1, TYPE_II = 2, TYPE_III = 3,
                   TYPE_IV = 4, TYPE_V = 5, TYPE_VI = 6, TYPE_VII = 7,
                   TYPE_VIII = 8, TYPE_IX = 9, TYPE_X = 10 };
typedef enum _enum_type_        enum_type_;


/* search prob. info */
struct _problem_ {
        int     domain;
        int     size;           /* problem size */
        int     seed;           /* random seed */
        domain  opt_value;      /* the optimal value */
        domain  huge;           /* huge number in the domain */
        task_   task;           /* task: search or learning */
};
typedef struct _problem_        problem_;


struct _constr_ {
```

```
        long    time;           /* time constraint */
        long    space;          /* space constraint */
        float   cst;            /* cst constraint */
        domain  bound;          /* bound of sol val, useful in ida threshold */
};
typedef struct _constr_         constr_;


struct _profile_ {
        yesno_  rt;
        yesno_  st;
        float   last_approx;
        float   last_accu;
        float   step;           /* run-time profile step */
};
typedef struct _profile_        profile_;


struct _factor_ {
        float   g;              /* stca, stcgd */
        float   s;              /* ptca, dtca, ptcgd, dtcgd */
        float   c;              /* ptca, dtca, ptcgd, dtcgd */
};
typedef struct _factor_         factor_;


typedef enum { FIXED_BW, LINEAR_BW, EXP_BW } bw_fx_;


struct _cmd_ {
        int     num_pe;         /* num of processing elements */
        int     comm_idle;      /* comm overhead in terms of num_op */
        int     first;          /* style of starting search */
        int     iter;           /* num of sample runs */
        int     debug;          /* debugging mode */
        float   cut_ratio;      /* cut-line ratio */
        yesno_  xon;            /* accuracy is on */
        yesno_  report;         /* special report, eg ida */
        count_  count;          /* counting system */
        strategy_ strategy;     /* search strategy */
        algorithm_ algorithm;   /* algorithm */
        float   approx;         /* hard approx */
        float   accu;           /* hard accuracy */
        constr_ constr;         /* hard constr */
        metric  bound_metric;   /** relative or absolute **/
        domain  bound_base;     /** relative base, usually opt sol val **/
        profile_ pf;            /* profile command */
        factor_ factor;         /* parameters for algorithm */
        bw_fx_  bw_fx;          /** bandwidth function name **/
        int     nparam;         /* num of parameters */
        char    **param;        /* parameter list */
};
typedef struct _cmd_            cmd_;


struct _io_ {
        char    *stat;          /* stat file for learning */
        char    *summary;       /* summary of statistics */
        char    *graph;         /* graph ready format */
        char    *rt_pf;         /* run-time profile */
        char    *st_pf;         /* space-vs-time profile */
        char    *report;        /* special report, eg ida */
};
typedef struct _io_             io_;
```

```
struct _stat_ {
        long    num_op;         /* # operations in parallel search */
        long    generated;      /* # nodes generated */
        long    expanded;       /* # nodes expanded */
        long    feasible;       /* # nodes feasible */
        long    infeasible;     /* # nodes infeasible */
        long    active;         /* # nodes active */
        long    bounded;        /* # nodes bounded */
        long    dominated;      /* # nodes dominated */
        long    bounding;       /* # nodes bounded by new node */
        long    dominating;     /* # nodes dominated by new node */
        long    hard_bounded;   /* # nodes hard bounded, for ida */
        /* pruned == bounded + dominated + bounding + dominating */
        long    time;           /* real timer */
        long    max_active;     /* max space usage */
        float   v_cst;          /* virtual cst */
        float   r_cst;          /* real cst */
        long    unix_utime;     /* user time provided by unix */
        long    unix_stime;     /* system time provided by unix */
};
typedef struct _stat_           stat_;


struct _approx_ {
        float   approx;
        float   accu;
        float   run_time;
        float   predicted;
        float   root_approx;
        float   root_accu;
        float   x_root_approx;  /* external root's approx degree */
        float   achieved;       /* approx achieved already */
};
typedef struct _approx_         approx_;


/*
        NODE STRUCTURE node_
        a node structure node_ consists of three parts:
        (a) PI (problem independent):
            permanent and predefined in B&B shell.
        (b) PDSI (problem independent but size dependent):
            user-defined in 'node.h'
        (c) PDSD (problem and size dependent):
            user-defined in 'node.h' with an array of sizes for all regions.
            all PDSD regions are accessed via pointers in PDSI.
*/

enum _nodetype_ { INACTIVE, ACTIVE_SINGLE, ACTIVE_COMPOUND };
typedef enum _nodetype_         nodetype_;


#define PDSI_   struct  PDSI__
#define PDSD_   struct  PDSD__


struct _node_ {
        struct _node_   *parent;
        struct _node_   *next;          /* used in the search tree */
        struct _node_   *brother;       /* used in children list */

        struct _node_   *left;          /* used in b_plus tree */
        struct _node_   *right;         /* used in b_plus tree */
        struct _node_   *up;            /* used in b_plus tree */

        int     entity;         /* entity id, ie, city id currently visited */
        int     depth;          /* depth of this node */
        int     nsprout;        /* number of live children */
        int     ndecomp;        /* number of docomposed */
        nodetype_ type;         /* type of search node,
                                   non-zero active, 0 inactive;
                                   specifically, 2: compound; 1: ordinary */
        domain  g_cost;         /* g_cost is the current node's actual cost */
        domain  lowb, upb;      /* node's lower bound and upper bound */

        PDSI_PART               /* PDSI part which is declared in 'node.h' */
};
typedef struct _node_           node_;


struct _node_conf_ {
        /* all sizes are in terms of integer */
        int     pdsd_size;      /* size of prob-dep, size-dep part */
        int     node_size;      /* size of a entire node */
        /* node_size == (pdsd_size + sizeof(node_)) */
};
typedef struct _node_conf_      node_conf_;


struct _regress_ {
        double  x_sum;  /* sum of x's */
        double  x2_sum; /* sum of x squares */
        double  y_sum;  /* sum of y's */
        double  xy_sum; /* sum of xy products */
        double  n;      /* number of points */
        double  beta[REGRESSION_ORDER + 1];     /* regression coeff's */
};
typedef struct _regress_        regress_;


struct _stca_ {
        float   g_factor;
};
typedef struct _stca_           stca_;


struct _stcgd_ {
        float   g_factor;
};
typedef struct _stcgd_          stcgd_;


struct _ptca_ {
        regress_        regress;
        float   s_factor;
        float   c_factor;
};
typedef struct _ptca_           ptca_;


struct _ptcgd_ {
        regress_        regress;
        float   g_factor;
        float   s_factor;
        float   c_factor;
};
```

```
typedef struct _ptcgd_          ptcgd_;


struct _dtca_ {
        regress_        regress;
        float   s_factor;
        float   c_factor;
};
typedef struct _dtca_           dtca_;


struct _myida_ {
        domain  threshold;
        float   g_factor;
};
typedef struct _myida_          myida_;


struct _ida_ {
        float   ida_g_fact;
        int     ida_n;                  /* Number of histogram bin */
        int     *ida_bin;               /* Histogram bin */
};
typedef struct _ida_            ida_;


struct _ida_list_ {
        struct _ida_list_       *next;
        int                     num_node;
        domain                  threshold;
};
typedef struct _ida_list_       ida_list_;


struct _ida_stat_ {
        int     iter;
        int     prev_node_expanded;
        float   max_exp_factor;
        float   tot_exp_factor;
        float   min_exp_factor;
        domain  threshold;
        ida_list_       *ida_hist;
};
typedef struct _ida_stat_       ida_stat_;


enum _unary_alg_ { UNARY_APPROX = 1, UNARY_TH = 2 };
typedef enum _unary_alg_        unary_alg_;


enum _th_alg_ { NAIVE_TH = 1, STATIC_TH = 2, PREDICTIVE_TH = 3 };
typedef enum _th_alg_           th_alg_;


enum _iter_attr_ { LAST_ITER = 1, LAST_2ND_ITER = 2, OTHER_ITER = 3 };
typedef enum _iter_attr_        iter_attr_;


enum _span_ { FULL_SPAN = 1, PARTIAL_SPAN = 2 };
typedef enum _span_             span_;


struct _set_th_ {
        th_alg_         alg;
        enum_type_      type;
        int             nbin;
        long            *bin;
        domain          *calibrate;
};
typedef struct _set_th_         set_th_;


struct _urts_ {
        yesno_          last_pred;
        domain          th;
        enum_type_      a_type;         /* approx type */
        float           g_factor;       /* step factor */
        float           r_factor;       /* growth rate */
        unary_alg_      alg;
        set_th_         *set_th;
};
typedef struct _urts_           urts_;


struct _brts_ {
        yesno_          last_pred;
        domain          th;
        enum_type_      a_type;         /* approx type */
        float           g_factor;       /* step factor */
        float           r_factor;       /* growth rate */
        set_th_         *set_th;
};
typedef struct _brts_           brts_;


struct _hrts_ {
        yesno_          last_pred;
        domain          th;
        enum_type_      a_type;         /* approx type */
        float           g_factor;       /* step factor */
        float           r_factor;       /* growth rate */
        set_th_         *set_th;
};
typedef struct _hrts_           hrts_;


struct _ird_basic_ {
        yesno_          shortcut;       /* jump if upper not change too much */
        float           base;           /* starting approx or threshold */
        float           margin;         /* shortcut margin */
};
typedef struct _ird_basic_      ird_basic_;


struct _uird_ {
        ird_basic_      basic;
        domain          th;
        enum_type_      a_type;         /* approx type */
        set_th_         *set_th;
        float           g_factor;       /* step factor */
        float           r_factor;       /* growth rate */
        unary_alg_      alg;
};
typedef struct _uird_           uird_;


struct _bird_ {
        ird_basic_      basic;
        domain          th;
        enum_type_      a_type;         /* approx type */
        set_th_         *set_th;
        float           g_factor;       /* step factor */
        float           r_factor;       /* growth rate */
};
typedef struct _bird_           bird_;


struct _search_message_ {
        search_style_   style;
        regress_        *rp;
        yesno_          alg_entry;
        yesno_          rt_pf;
        yesno_          st_pf;
        yesno_          feasible;
};
typedef struct _search_message_ search_message_;


typedef struct {
        node_           *young;         /** list of youngs of this depth **/
        int             count;          /** num of adults of this depth **/
        int             width;          /** width of this depth **/
} tree_;


typedef struct {
        int             ceiling_depth;  /** lower bound of depth **/
        int             floor_depth;    /** upper bound of depth **/
        int             init_width;     /** init_width of band **/
        tree_           *tree;          /** band search's array of youngs **/
} band_;


struct _search_
{
        strategy_       strategy;
        algorithm_      algorithm;
        domain          glub;           /* global least upper bound */
        domain          gllb;           /* global least lower bound */
        domain          root_lb;        /* root's lower bound */
        domain          root_ub;        /* root's upper bound */
        domain          x_root_lb;      /* external lower bound */
        domain          x_root_ub;      /* external upper bound */
        stat_           stat;           /* statistics */
        constr_         constr;         /* constraints */
        approx_         approx;         /* approx */
        solution_       *incumbent;     /* incumbent */
        band_           band;           /** band descriptor in band search **/
        domain          next_ida_th;    /** for IDA* only **/
        union active_list_
        {
          node_         *bptree;        /* bfs b_plus tree */
          node_         *btree;         /* bfs virtual b+ tree, & band search */
          node_         *list;          /* open list, for gdfs */
          node_         *stack;         /* activation stack, for dfs */
        } open;
        union alg_struct_
        {
          void          *def;           /* no associated structure is needed */
          void          *lw;
          stca_         *stca;
          ptca_         *ptca;
          dtca_         *dtca;
          stcgd_        *stcgd;
          ptcgd_        *ptcgd;
          urts_         *urts;
          brts_         *brts;
          hrts_         *hrts;
          uird_         *uird;
          bird_         *bird;
        } alg;
        struct _search_ *child;         /* any sub-search */
        struct _search_ *brother;       /* search of a level */
        struct _search_ *parent;        /* parent search */
        int             idle_clk;       /* idle timer for parallel processing */
};
typedef struct _search_         search_;


#endif __wise_define_h_

::::::::::::::
include/var.c
::::::::::::::

problem_        problem;
cmd_            cmd;
io_             io;
node_           *pool_manager = NULL, *savep = NULL, *ROOT = NULL;
node_conf_      node_conf;
struct tms      start_time, last_time, break_time;

int             SIZE = 0;       /** for HARE **/
int             huge_int = HUGE_INT;
long            huge_long = HUGE_LONG;
float           huge_float = HUGE_FLOAT;
double          huge_double = HUGE_DOUBLE;

long            MaxDepth = FirstMaxDepth;
long            NumNodesTaken = 0;
int             RTS_IRD_iter = 0;
domain          Next_IDA_Threshold = (domain) 0;
yesno_          No_Upper_Bound = NO;

stat_           keep_stat[MaxNumberOfProblems];


void init_problem_struct (p)
  problem_ *p;
{
  switch (p->domain = Domain) {
    case INT:    p->huge = HUGE_INT;    break;
    case LONG:   p->huge = HUGE_LONG;   break;
    case FLOAT:  p->huge = HUGE_FLOAT;  break;
    case DOUBLE: p->huge = HUGE_DOUBLE; break;
    default:     error("no such generic data type\n"); break;
  }
  p->opt_value = p->huge;
  p->seed = 1;
  p->task = CSP;
}


void init_cmd_struct (p)
  cmd_ *p;
{
  p->num_pe = 1;
  p->comm_idle = 0;
  p->first = 0;
  p->iter = 1;
  p->debug = 0;
  p->cut_ratio = 0.5;
  p->xon = NO;
  p->report = NO;
  p->count = VIRTUAL;
  p->strategy = BFS;
  p->algorithm = DEFAULT;
  p->approx = 0.0;
  p->accu = 1.0;
  p->constr.time = HUGE_LONG;
  p->constr.space = HUGE_LONG;
  p->constr.cst = HUGE_FLOAT;
  p->constr.bound = problem.huge;
  p->bound_metric = ABSOLUTE;
  p->bound_base = (domain) problem.huge;
  init_profile_struct(&(p->pf));
  p->bw_fx = FIXED_BW;
  p->nparam = 0;
  p->param = NULL;
}


void init_factor_struct (p)
  factor_ *p;
{
  p->g = 0.25;
  p->s = 0.15;
  p->c = 0.8;
}


void init_io_struct (iop)
  io_ *iop;
{
  iop->stat = "stat";             /* old out */
  iop->summary = "summary";       /* old constr.out */
  iop->graph = "graph-ready";
  iop->rt_pf = "run-time";
  iop->st_pf = "space-time";
  iop->report = "report";
}


void init_stat_struct (p)
  stat_ *p;
{
  p->num_op = 0;
  p->generated = 0;
  p->expanded = 0;
  p->feasible = 0;
  p->infeasible = 0;
  p->active = 0;
  p->bounded = 0;
  p->dominated = 0;
  p->bounding = 0;
  p->dominating = 0;
  p->hard_bounded = 0;
  p->max_active = 0;
  p->time = 0;
  p->v_cst = 0.0;
```

```
  p->r_cst = 0.0;
  p->unix_utime = 0;
  p->unix_stime = 0;
}


void init_constr_struct (p)
  constr_ *p;
{
  p->time = cmd.constr.time;
  p->space = cmd.constr.space;
  p->cst = cmd.constr.cst;
  p->bound = cmd.constr.bound;
}


void init_profile_struct (p)
  profile_ *p;
{
  p->rt = NO;
  p->st = NO;
  p->step = 0.01;
  p->last_approx = HUGE_FLOAT;
  p->last_accu = 0.0;
}


void init_approx_struct (p)
  approx_ *p;
{
  p->approx = cmd.approx;
  p->accu = cmd.accu;
  p->predicted = 0.0;
  p->run_time= HUGE_FLOAT;
  p->root_approx = HUGE_FLOAT;
  p->x_root_approx = HUGE_FLOAT;
  p->achieved = HUGE_FLOAT;
}


void init_band_struct (sp)
    search_ *sp;
{   band_ *bp = &(sp->band);
    int i, num;

    bp->ceiling_depth = 0;
    bp->floor_depth = 0;
    bp->init_width = get_int_cmd_line_param (HUGE_INT, 0, 0);

    if (cmd.nparam <= 1) cmd.bw_fx = FIXED_BW;
    else {
        switch (**(cmd.param+1)) {
        case 'l':
        case 'L': cmd.bw_fx = LINEAR_BW; break;
        case 'e':
        case 'E': cmd.bw_fx = EXP_BW; break;
        case 'f':
        case 'F':
        case '-':
        default:  cmd.bw_fx = FIXED_BW; break;
        }
    }

    if (sp->strategy == BAND) {
        MaxDepth = FirstMaxDepth;
        num = MaxDepth + 1;
        sp->band.tree = (tree_ *) malloc (num * sizeof (tree_));
        for (i = 0; i < num; i++)
            init_tree_struct (sp->band.tree, i, sp->band.init_width);
    }
    else sp->band.tree = NULL;
}


void init_tree_struct (tp, depth, init_width)
    tree_ *tp;
    int depth, init_width;
{
    (tp+depth)->young = NULL;
    (tp+depth)->count = 0;
    (tp+depth)->width = bandwidth_function (depth, init_width);
}


void init_search_struct (sp, init_open_ptr, init_alg_ptr)
  search_ *sp;
  int init_open_ptr, init_alg_ptr;
{
  sp->strategy = cmd.strategy;
  sp->algorithm = cmd.algorithm;
  init_stat_struct(&(sp->stat));
  init_constr_struct(&(sp->constr));
  init_approx_struct(&(sp->approx));
  if (init_open_ptr) sp->open.bptree = NULL;
  if (init_alg_ptr) sp->alg.def = NULL;
  sp->child = NULL;
  sp->brother = NULL;
  sp->parent = NULL;
  sp->idle_clk = 0;
  init_band_struct (sp);
}


void init_regress_struct (rp)
  regress_ *rp;
{ int i;

  rp->x_sum = 0.0;
  rp->x2_sum = 0.0;
  rp->y_sum = 0.0;
  rp->xy_sum = 0.0;
  rp->n = 0.0;
  for (i = 0; i <= REGRESSION_ORDER; i++) rp->beta[i] = 0.0;
}


void init_stca_struct (p)
  stca_ *p;
{ p->g_factor = cmd.factor.g; }


void init_ptca_struct (p)
  ptca_ *p;
{
  p->s_factor = cmd.factor.s;
  p->c_factor = cmd.factor.c;
  init_regress_struct(&(p->regress));
}
```

```
void init_dtca_struct (p)
  dtca_ *p;
{
  p->s_factor = cmd.factor.s;
  p->c_factor = cmd.factor.c;
  init_regress_struct(&(p->regress));
}


void init_stcgd_struct (p)
  stcgd_ *p;
{ p->g_factor = cmd.factor.g; }


void init_ptcgd_struct (p)
  ptcgd_ *p;
{
  p->g_factor = cmd.factor.g;
  p->s_factor = cmd.factor.s;
  p->c_factor = cmd.factor.c;
  init_regress_struct(&(p->regress));
}


void init_set_th_struct (p, idx)
  set_th_ *p;
  int idx;
{
  switch (**(cmd.param + idx++)) {
    case '-':
    case 'n': p->alg = NAIVE_TH;        break;
    case 's': p->alg = STATIC_TH;       break;
    case 'p': p->alg = PREDICTIVE_TH;   break;
    default: error("init_set_th_struct: no such algorithm"); break;
  }

  switch (**(cmd.param + idx)) {
    case '-':
    case '1': p->type = TYPE_I;   break;
    case '2': p->type = TYPE_II;  break;
    case '3': p->type = TYPE_III; break;
    case '4': p->type = TYPE_IV; break;
    default: error("init_set_th_struct: no such type"); break;
  }

  p->nbin = NumberOfBins;
  p->bin = NULL;
  p->calibrate = NULL;
}


void create_th_bin (set_th)
  set_th_ *set_th;
{ int i;

  set_th->bin = (long *) malloc(set_th->nbin * sizeof(long));
  set_th->calibrate = (domain *) malloc(set_th->nbin * sizeof(domain));
  for (i = 0; i < set_th->nbin; i++) {
    *(set_th->bin + i) = (long) 0;
    *(set_th->calibrate + i) = (domain) 0;
  }
}
```

```
float get_float_cmd_line_param (def_val, loc, offset)
  float def_val;
  int loc, offset;
{ float val;

  if (loc >= cmd.nparam) return def_val;
  if (*(*(cmd.param + loc) + offset) == '-') return def_val;
  sscanf((*(cmd.param + loc) + offset), "%f", &val);
  return val;
}


int get_int_cmd_line_param (def_val, loc, offset)
  int def_val;
  int loc, offset;
{ int val;

  if (loc >= cmd.nparam) return def_val;
  if (*(*(cmd.param + loc) + offset) == '-') return def_val;
  sscanf((*(cmd.param + loc) + offset), "%d", &val);
  return val;
}


void init_urts_struct (p)
  urts_ *p;
{
  p->last_pred = (**(cmd.param) == 'y') ? YES : NO;
  p->th = problem.huge;
  switch (*(*(cmd.param+1) + 1)) {
    case '1': p->a_type = TYPE_I;   break;
    case '2': p->a_type = TYPE_II;  break;
    case '3': p->a_type = TYPE_III; break;
    case '4': p->a_type = TYPE_IV;  break;
    default:  p->a_type = TYPE_I;   break;
  }
  p->g_factor = get_float_cmd_line_param (HUGE_FLOAT, 4, 0);
  p->r_factor = get_float_cmd_line_param (HUGE_FLOAT, 5, 0);
  p->alg = (**(cmd.param+1) == 'a') ? UNARY_APPROX : UNARY_TH;
  p->set_th = (set_th_ *) malloc (sizeof (set_th_));
  init_set_th_struct (p->set_th, 2);
  if (cmd.nparam >= 7) p->set_th->nbin = atoi (*(cmd.param+6));
  create_th_bin (p->set_th);
}


void init_brts_struct (p)
  brts_ *p;
{
  p->last_pred = (**(cmd.param) == 'y') ? YES : NO;
  p->th = problem.huge;
  switch (*(*(cmd.param+1) + 1)) {
    case '1': p->a_type = TYPE_I;   break;
    case '2': p->a_type = TYPE_II;  break;
    case '3': p->a_type = TYPE_III; break;
    case '4': p->a_type = TYPE_IV;  break;
    default:  p->a_type = TYPE_I;   break;
  }
  p->g_factor = get_float_cmd_line_param(HUGE_FLOAT, 3, 0);
  p->r_factor = get_float_cmd_line_param(HUGE_FLOAT, 4, 0);
  p->set_th = (set_th_ *) malloc(sizeof(set_th_));
  init_set_th_struct(p->set_th, 1);
  if (cmd.nparam >= 6) p->set_th->nbin = atoi(*(cmd.param + 5));
  create_th_bin(p->set_th);
}


void init_hrts_struct (p)
  hrts_ *p;
{
  p->last_pred = (**(cmd.param) == 'y') ? YES : NO;
  p->th = problem.huge;
  switch (*(*(cmd.param+1) + 1)) {
    case '1': p->a_type = TYPE_I;   break;
    case '2': p->a_type = TYPE_II;  break;
    case '3': p->a_type = TYPE_III; break;
    case '4': p->a_type = TYPE_IV;  break;
    default:  p->a_type = TYPE_I;   break;
  }
  p->g_factor = get_float_cmd_line_param(HUGE_FLOAT, 3, 0);
  p->r_factor = get_float_cmd_line_param(HUGE_FLOAT, 4, 0);
  p->set_th = (set_th_ *) malloc(sizeof(set_th_));
  init_set_th_struct(p->set_th, 1);
  if (cmd.nparam >= 6) p->set_th->nbin = atoi(*(cmd.param + 5));
  create_th_bin(p->set_th);
}


void init_uird_struct (p)
  uird_ *p;
{
  if ((p->basic.shortcut = (**(cmd.param) == 'y') ? YES : NO) == YES)
    p->basic.margin = get_float_cmd_line_param(0.0, 0, 1);
  p->th = problem.huge;
  switch (*(*(cmd.param+1) + 1)) {
    case '1': p->a_type = TYPE_I;   break;
    case '2': p->a_type = TYPE_II;  break;
    case '3': p->a_type = TYPE_III; break;
    case '4': p->a_type = TYPE_IV;  break;
    default:  p->a_type = TYPE_I;   break;
  }
  p->g_factor = get_float_cmd_line_param(HUGE_FLOAT, 4, 0);
  p->r_factor = get_float_cmd_line_param(HUGE_FLOAT, 5, 0);
  p->basic.base = get_float_cmd_line_param(0.0, 6, 0);
  p->alg = (**(cmd.param+1) == 'a') ? UNARY_APPROX : UNARY_TH;
  p->set_th = (set_th_ *) malloc(sizeof(set_th_));
  init_set_th_struct(p->set_th, 2);
  if (cmd.nparam >= 8) p->set_th->nbin = atoi(*(cmd.param + 7));
  create_th_bin(p->set_th);
}


void init_bird_struct (p)
  bird_ *p;
{
  if ((p->basic.shortcut = (**(cmd.param) == 'y') ? YES : NO) == YES)
    p->basic.margin = get_float_cmd_line_param(0.0, 0, 1);
  p->th = problem.huge;
  switch (*(*(cmd.param+1) + 1)) {
    case '1': p->a_type = TYPE_I;   break;
    case '2': p->a_type = TYPE_II;  break;
    case '3': p->a_type = TYPE_III; break;
    case '4': p->a_type = TYPE_IV;  break;
    default:  p->a_type = TYPE_I;   break;
  }
  p->g_factor = get_float_cmd_line_param(HUGE_FLOAT, 3, 0);
  p->r_factor = get_float_cmd_line_param(HUGE_FLOAT, 4, 0);
  p->basic.base = get_float_cmd_line_param(0.0, 5, 0);
  p->set_th = (set_th_ *) malloc(sizeof(set_th_));
  init_set_th_struct(p->set_th, 1);
  if (cmd.nparam >= 7) p->set_th->nbin = atoi(*(cmd.param + 6));
  create_th_bin(p->set_th);
}


void init_ida_struct (p)
  ida_ *p;
{
  p->ida_g_fact = 3;
  p->ida_n = 20;
  p->ida_bin = NULL;
}


void init_node_struct (p, parentp)
  node_ *p, *parentp;
{
  p->parent = parentp;
  p->next = p->brother = NULL;
  p->left = p->right = p->up = NULL;
  p->nsprout = 0;
  p->type = 1;
  p->ndecomp = 0;
  if (parentp) { (parentp->nsprout)++; p->depth = parentp->depth + 1; }
  else p->depth = 0;
}


::::::::::::::
include/var.h
::::::::::::::

#ifndef __wise_var_h_
#define __wise_var_h_

extern problem_   problem;
extern cmd_       cmd;
extern io_        io;
extern node_      *pool_manager, *savep, *ROOT;
extern node_conf_ node_conf;
extern struct tms start_time, last_time, break_time;

extern int        SIZE;
extern int        huge_int;
extern long       huge_long;
extern float      huge_float;
extern double     huge_double;

extern long       MaxDepth;
extern long       NumNodesTaken;
extern int        RTS_IRD_iter;
extern domain     Next_IDA_Threshold;
extern yesno_     No_Upper_Bound;

extern stat_      keep_stat[MaxNumberOfProblems];
#endif __wise_var_h_

::::::::::::::
include/normal.c
::::::::::::::
```

```
/** cdf table for the standard normal distribution,
 ** each entry differs by 0.05, starting from 0.00 to 3.00, inclusively.
 **/
snormal_cdf[61] = {
/** 0.00 **/  .500,     /** 0.05 **/  .520,     /** 0.10 **/  .540,
/** 0.15 **/  .560,     /** 0.20 **/  .579,     /** 0.25 **/  .599,
/** 0.30 **/  .618,     /** 0.35 **/  .637,     /** 0.40 **/  .655,
/** 0.45 **/  .674,     /** 0.50 **/  .691,     /** 0.55 **/  .709,
/** 0.60 **/  .726,     /** 0.65 **/  .742,     /** 0.70 **/  .758,
/** 0.75 **/  .773,     /** 0.80 **/  .788,     /** 0.85 **/  .802,
/** 0.90 **/  .816,     /** 0.95 **/  .829,     /** 1.00 **/  .841,
/** 1.05 **/  .853,     /** 1.10 **/  .864,     /** 1.15 **/  .875,
/** 1.20 **/  .885,     /** 1.25 **/  .894,     /** 1.30 **/  .903,
/** 1.35 **/  .911,     /** 1.40 **/  .919,     /** 1.45 **/  .926,
/** 1.50 **/  .933,     /** 1.55 **/  .939,     /** 1.60 **/  .945,
/** 1.65 **/  .951,     /** 1.70 **/  .955,     /** 1.75 **/  .960,
/** 1.80 **/  .964,     /** 1.85 **/  .968,     /** 1.90 **/  .971,
/** 1.95 **/  .974,     /** 2.00 **/  .977,     /** 2.05 **/  .980,
/** 2.10 **/  .982,     /** 2.15 **/  .984,     /** 2.20 **/  .986,
/** 2.25 **/  .988,     /** 2.30 **/  .989,     /** 2.35 **/  .991,
/** 2.40 **/  .992,     /** 2.45 **/  .993,     /** 2.50 **/  .994,
/** 2.55 **/  .995,     /** 2.60 **/  .995,     /** 2.65 **/  .996,
/** 2.70 **/  .997,     /** 2.75 **/  .997,     /** 2.80 **/  .997,
/** 2.85 **/  .998,     /** 2.90 **/  .998,     /** 2.95 **/  .998,
/** 3.00 **/  .999
};
::::::::::::::
include/debug.h
::::::::::::::

#ifdef DEBUG
#ifndef __wise_debug_h_
#define __wise_debug_h_

extern char *dbg_task[2];
extern char *dbg_datatype[5];
extern char *dbg_strategy[4];
extern char *dbg_algorithm[10];
extern char *dbg_count[2];
extern char *dbg_bw_fx[3];

#endif __wise_debug_h_
#endif DEBUG

::::::::::::::
include/debug.c
::::::::::::::

#ifdef DEBUG
#ifndef _csp_debug_c_
#define _csp_debug_c_

char *dbg_task[] = { "LEARNING" , "CSP" };
char *dbg_datatype[] = { "SHORT", "INT", "LONG", "FLOAT", "DOUBLE" };
char *dbg_strategy[] = { "GBB", "BFS", "DFS", "GDFS", "BAND" };
char *dbg_algorithm[] = { "DEFAULT", "LW", "sTCA", "pTCA", "dTCA", "sTCGD",
                          "pTCGD", "uRTS", "bRTS", "hRTS", "uIRD", "bIRD",
                          "IRA", "IDA", "DFS_star", "PBFS", "PDFS", "PGDFS", "PGBB" };
char *dbg_count[] = { "REAL", "VIRTUAL" };
char *dbg_bw_fx[] = { "FIXED_BW", "LINEAR_BW", "EXP_BW" };

#endif _csp_debug_c_
#endif DEBUG
::::::::::::::
```

```
include/config.h
::::::::::::::

#ifndef __wise_config_h_
#define __wise_config_h_

/* kernel/etc.c */
extern int        ceiling(), is_any(), is_member(), is_invalid_gllb(),
                  is_compound(), is_algo_rt_pf(), gen_random_int();
extern float      gen_random_float(), gen_random_range();
extern solution_  *get_sol_buf();
extern void       error(), set_compound(), free_sol_buf(), set_random_radix(),
                  set_search_task(), debug_node(), set_node_size(), junk();

/* kernel/free.c */
extern void       free_environ(), free_alg_struct(), free_open(), free_list(),
                  free_btree(), free_stack(), free_node();
extern long       free_bptree ();

/* kernel/init.c */
extern node_      *create_root();
extern void       clear_files(), attach_alg_struct(), init_alg_struct(),
                  examine_root();
extern search_    *init_environ();

/* kernel/limit.c */
extern int        is_constr_violated();
extern float      calc_rt_approx(), get_rt_approx(), get_approx();
extern domain     expected_opt(), get_threshold();
extern yesno_     is_threshold_related();
extern void       eval_rt_approx(), put_achieved_approx(), eval_final_approx(),
                  put_threshold(), merge_threshold_to_parent(),
                  put_child_constr(), set_stop_constr();

/* kernel/para.c */
extern yesno_     para_termination ();
extern void       para_merge_solution (), para_merge_stat (), para_load_balancing ();
extern node_      *para_request ();

/* kernel/profile.c */
extern int        is_in_transient_phase();
extern void       pf_run_time(), pf_space_vs_time(), alg_rt_pf(), alg_pf(),
                  alg_pf_predict();

/* kernel/search.c */
extern int        is_bounded(), is_hard_bounded();
extern search_    *create_brother_search(), *create_child_search();
extern void       search(), alg_entry(), default_search(), dominating(),
                  bounding(), btree_bounding(), list_bounding(),
                  bptree_bounding(), set_search_message();

/* kernel/stat.c */
extern long       get_time(), get_space(), get_max_space(), get_pruned(),
                  get_offset_time(), get_offset_space(), get_offset_max_space(),
                  get_offset_gen();
extern float      get_cst(), get_offset_rt_approx(), get_offset_cst();
extern void       update_stat(), merge_stat_to_parent(), merge_approx_to_parent(),
                  merge_sol_to_parent(), inherit_sol_from_parent();

/* include/var.c */
extern float      get_float_cmd_line_param();
extern void       init_env_struct(), init_problem_struct(), init_cmd_struct(),
                  init_factor_struct(), init_io_struct(), init_stat_struct(),
                  init_constr_struct(), init_profile_struct(),
                  init_band_struct (), init_tree_struct (),
                  init_approx_struct(), init_search_struct(),
                  init_regress_struct(), init_stca_struct(), init_ptca_struct(),
                  init_dtca_struct(), init_stcgd_struct(), init_ptcgd_struct(),
                  init_dtcgd_struct(), init_ida_struct(), init_node_struct(),
                  init_set_th_struct(), create_th_bin(), init_urts_struct(),
                  init_brts_struct(), init_hrts_struct();

/* primitve/bfs.c */
extern search_ending_ bfs_primitive();

/* primitve/gbb.c */
extern search_ending_ gbb_primitive();

/* primitive/dfs.c */
extern search_ending_ dfs_primitive();

/* primitive/gdfs.c */
extern search_ending_ gdfs_primitive();

/* primitive/path.c */
extern search_ending_ path_primitive();

/* primitive/first.c */
extern search_ending_ first_primitive();

/* algorithm/default.c */
extern void       default_algorithm();

/* algorithm/lw.c */
extern yesno_     is_lw_end();
extern void       lawler_wood_algorithm(), lw_set_constr();

/* algorithm/para.c */
extern void       para_algorithm ();
extern yesno_     para_primitive ();

/* algorithm/tca.c */
extern float      get_stca_next_approx();
extern void       stca_algorithm(), ptca_algorithm(), dtca_algorithm(),
                  dtca_entry();

/* algorithm/tcgd.c */
extern float      get_stcgd_next_approx();
extern void       stcgd_algorithm(), ptcgd_algorithm();

/* algorithm/rts.c */
extern void       urts_algorithm(), urts_entry(),
                  brts_algorithm(), brts_entry(),
                  hrts_algorithm(), hrts_entry();

/* algorithm/ird.c */
extern void       uird_algorithm(), uird_entry(),
                  bird_algorithm(), bird_entry();

/** open/bandlist.c **/
extern node_      *get_band_node (), *young_delete ();
extern void       put_band_node (), young_insert ();
extern int        is_young_empty ();

/* open/bptree.c */
extern int        is_bptree_empty();
extern node_      *bptree_delete();
extern void       bptree_insert();
```

```
/* open/btree.c */
extern int      is_btree_empty();
extern node_    *btree_delete();
extern void     btree_insert();

/* open/list.c */
extern int      is_list_empty();
extern node_    *list_delete();
extern void     list_insert();

/* open/pool.c */
extern node_    *get_search_node_from_pool();
extern void     dispose_search_node_to_pool();

/* open/sort.c */
extern int      is_open_empty();
extern node_    *delete();
extern domain   sort_key();
extern void     insert();

/* open/stack.c */
extern int      is_stack_empty();
extern node_    *stack_top(), *stack_bottom();
extern void     stack_push(), stack_pop();

/* interface/command.c */
extern void     cmd_line();

/* interface/output.c */
extern void     flush_accounting(), flush_summary(), flush_graph();

/* interface/result.c */
extern void     keep_result(), print_result();

#endif __wise_config_h_
```

```
Thu Jan 30 15:19:21 CST 1992
::::::::::::::
interface/command.c
::::::::::::::
/* begin, shorthand */
#define is_not_default          (*(argv[index]) != '-')
#define inc_index(abort_cond)   if (++index >= argc) { \
                                  if (abort_cond) error(errmsg); \
                                  else break; \
                                } \
                                if (*(argv[index]) == ':') \
                                  if (abort_cond) error(errmsg); \
                                  else continue;
/* end, shorthand */


void cmd_line (argc, argv)
  int argc;
  char *argv[];
{ int index = 1, is_prob_spec = 0, i;
  enum_type_ t1 = TYPE_NULL, t2 = TYPE_NULL;
  float f_val;
  char *errmsg = "not sufficient arguments\n";
#ifdef DEBUG
if (cmd.debug >= 1) {
  printf("enter cmd_line processor\n");
  for(index = 0; index < argc; index++) printf("%s ", argv[index]);
  printf("\n");
  index = 1;
}
#endif

  if (argc <= 1) error(errmsg);

  while (index < argc) {

    if (streql(argv[index],":real")) {
      cmd.count = REAL;
      inc_index(0);
    } /* :real */

    else if (streql(argv[index],":virtual")) {
      cmd.count = VIRTUAL;
      inc_index(0);
    } /* :virtual */

    else if (streql(argv[index],":task")) {
      inc_index(1);
      if (streql(argv[index],"csp")) problem.task = CSP;
      else if (streql(argv[index],"learn")) problem.task = LEARNING;
      else error("no such search task\n");
      inc_index(0);
    }

    else if (streql(argv[index],":report")) {
      cmd.report = 1;
      inc_index(0);
      if (is_not_default) io.report = argv[index];
      inc_index(0);
    } /* :report */

    else if (streql(argv[index],":search")) {
      inc_index(1);
      if (streql(argv[index],"bfs")) cmd.strategy = BFS;
```

```
      else if (streql(argv[index],"dfs")) cmd.strategy = DFS;
      else if (streql(argv[index],"gdfs")) cmd.strategy = GDFS;
      else if (streql(argv[index],"gbb")) cmd.strategy = GBB;
      else if (streql(argv[index],"band")) cmd.strategy = BAND;
      else error("no such search strategy\n");
      inc_index(0);

      /* TCA(approx) */
      if (streql(argv[index],"stca")) cmd.algorithm = sTCA;
      else if (streql(argv[index],"ptca")) cmd.algorithm = pTCA;
      else if (streql(argv[index],"dtca")) cmd.algorithm = dTCA;

      /* TCA(accu) */
      if (streql(argv[index],"sTCA"))
        { cmd.algorithm = sTCA; cmd.xon = YES; }
      else if (streql(argv[index],"pTCA"))
        { cmd.algorithm = pTCA; cmd.xon = YES; }
      else if (streql(argv[index],"dTCA"))
        { cmd.algorithm = dTCA; cmd.xon = YES; }

      /* TCGD(approx) */
      else if (streql(argv[index],"stcgd")) cmd.algorithm = sTCGD;
      else if (streql(argv[index],"ptcgd")) cmd.algorithm = pTCGD;

      /* TCGD(accu) */
      else if (streql(argv[index],"sTCGD"))
        { cmd.algorithm = sTCGD; cmd.xon = YES; }
      else if (streql(argv[index],"pTCGD"))
        { cmd.algorithm = pTCGD; cmd.xon = YES; }

      /* uRTS(approx) */
      else if (streql(argv[index],"urts")) cmd.algorithm = uRTS;

      /* uRTS(accu) */
      else if (streql(argv[index],"uRTS"))
        { cmd.algorithm = uRTS; cmd.xon = YES; }

      /* bRTS(approx) */
      else if (streql(argv[index],"brts")) cmd.algorithm = bRTS;

      /* bRTS(accu) */
      else if (streql(argv[index],"bRTS"))
        { cmd.algorithm = bRTS; cmd.xon = YES; }

      /* hRTS(approx) */
      else if (streql(argv[index],"hrts")) cmd.algorithm = hRTS;

      /* hRTS(accu) */
      else if (streql(argv[index],"hRTS"))
        { cmd.algorithm = hRTS; cmd.xon = YES; }

      /* uIRD(approx) */
      else if (streql (argv[index],"uird")) cmd.algorithm = uIRD;

      /* uIRD(accu) */
      else if (streql (argv[index],"uIRD"))
        { cmd.algorithm = uIRD; cmd.xon = YES; }

      /* bIRD(approx) */
      else if (streql (argv[index],"bird")) cmd.algorithm = bIRD;

      /* bIRD(accu) */
      else if (streql (argv[index],"bIRD"))
        { cmd.algorithm = bIRD; cmd.xon = YES; }
```

```
  /* LW(approx) */
  else if (streql (argv[index],"lw")) cmd.algorithm = LW;

  /* LW(accu) */
  else if (streql (argv[index],"LW"))
    { cmd.algorithm = LW; cmd.xon = YES; }

  /* IRA*(approx) */
  else if (streql (argv[index],"ira")) cmd.algorithm = IRA;

  /* IRA*(accu) */
  else if (streql (argv[index],"IRA"))
    { cmd.algorithm = IRA; cmd.xon = YES; }

  /* IDA*(approx) */
  else if (streql (argv[index],"ida")) cmd.algorithm = IDA;

  /* IDA*(accu) */
  else if (streql (argv[index],"IDA"))
    { cmd.algorithm = IDA; cmd.xon = YES; }

  /* DFS*(approx) */
  else if (streql (argv[index],"dfs")) cmd.algorithm = DFS_star;

  /* DFS*(accu) */
  else if (streql (argv[index],"DFS"))
    { cmd.algorithm = DFS_star; cmd.xon = YES; }

  /* PBFS(approx) */
  else if (streql (argv[index],"pbfs")) cmd.algorithm = PBFS;

  /* PBAND(approx) */
  else if (streql (argv[index],"pband")) cmd.algorithm = PBAND;

  /* PBFS(accu) */
  else if (streql (argv[index],"PBFS"))
    { cmd.algorithm = PBFS; cmd.xon = YES; }

  /* PBAND(accu) */
  else if (streql (argv[index],"PBAND"))
    { cmd.algorithm = PBAND; cmd.xon = YES; }

  /* PDFS(approx) */
  else if (streql (argv[index],"pdfs")) cmd.algorithm = PDFS;

  /* PDFS(accu) */
  else if (streql (argv[index],"PDFS"))
    { cmd.algorithm = PDFS; cmd.xon = YES; }

  /* PGDFS(approx) */
  else if (streql (argv[index],"pgdfs")) cmd.algorithm = PGDFS;

  /* PGDFS(accu) */
  else if (streql (argv[index],"PGDFS"))
    { cmd.algorithm = PGDFS; cmd.xon = YES; }

  /* PGBB(approx) */
  else if (streql (argv[index],"pgbb")) cmd.algorithm = PGBB;

  /* PGBB(accu) */
  else if (streql (argv[index],"PGBB"))
    { cmd.algorithm = PGBB; cmd.xon = YES; }
```

```
  /* DEFAULT(approx) */
  else if (streql (argv[index],"def")) cmd.algorithm = DEFAULT;
  else if (streql (argv[index],"default")) cmd.algorithm = DEFAULT;

  /* DEFAULT(accu) */
  else if (streql (argv[index],"DEF"))
    { cmd.algorithm = DEFAULT; cmd.xon = YES; }
  else if (streql (argv[index],"DEFAULT"))
    { cmd.algorithm = DEFAULT; cmd.xon = YES; }

  else error("no such search algorithm\n");
  inc_index(0);
} /* :search */

else if (streql (argv[index],":prob")) {
  inc_index(1);
  cmd.iter = atoi (argv[index]);
  inc_index(1);
  problem.size = SIZE = atoi (argv[index]);
  inc_index(1)
  if (problem.size > MaxProblemSize) {
    printf("MaxProblemSize=%d, problem.size=%d\n",
            MaxProblemSize, problem.size);
    error("problem size too large\n");
  }
  problem.seed = atoi(argv[index]);
  is_prob_spec = 1;
  inc_index(0);
} /* :prob */

else if (streql(argv[index],":io")) {
  inc_index(1);
  if (is_not_default) io.stat = argv[index];
  inc_index(0);
  if (is_not_default) io.summary = argv[index];
  inc_index(0);
  if (is_not_default) io.graph = argv[index];
  inc_index(0);
  if (is_not_default) io.rt_pf = argv[index];
  inc_index(0);
  if (is_not_default) io.st_pf = argv[index];
  inc_index(0);
} /* :io */

else if (streql(argv[index],":approx")) {
  inc_index(1);
  if (is_not_default) sscanf(argv[index], "%f", &(cmd.approx));
  inc_index(0);
} /* :approx */

else if (streql(argv[index],":accu")) {
  inc_index(1);
  if (is_not_default) sscanf(argv[index], "%f", &(cmd.accu));
  inc_index(0);
} /* :accu */

else if (streql(argv[index],":dbg")) {
  inc_index(1);
  if (is_not_default) cmd.debug = atoi(argv[index]);
  inc_index(0);
} /* :dbg */

else if (streql(argv[index],":constr")) {
  inc_index(1);
```

```
    if (is_not_default) cmd.constr.time = atoi(argv[index]);
    inc_index(0);
    if (is_not_default) cmd.constr.space = atoi(argv[index]);
    inc_index(0);
    if (is_not_default) {
      sscanf (argv[index], "%f", &f_val);
      cmd.constr.cst = f_val;
    }
    inc_index(0);
    if (is_not_default) {
      sscanf (argv[index], "%f", &f_val);
      cmd.constr.bound = (domain) f_val;
    }
    inc_index(0);
    if (is_not_default) {
      if (*(argv[index]) == 'r' || *(argv[index]) == 'R') {
        cmd.bound_metric = RELATIVE;
        if (isdigit (*(argv[index]+1)) || ispunct (*(argv[index]+1))) {
          sscanf (argv[index]+1, "%f", &f_val);
          cmd.bound_base = (domain) f_val;
        }
      }
    }
    inc_index(0);
  } /* :constr */

  else if (streql(argv[index],":pf")) {  /* profile */
    inc_index(1);
    if (is_not_default) {
      cmd.pf.rt = YES;
      sscanf(argv[index], "%f", &(cmd.pf.step));
    }
    inc_index(0);
    if (is_not_default) cmd.pf.st = YES;
    inc_index(0);
  } /* :pf */

  else if (streql (argv[index],":pe")) {      /* num of pe's */
    inc_index(1);
    if (is_not_default)
      cmd.num_pe = atoi (argv[index]);
    inc_index(0);
  } /* :pe */

  else if (streql (argv[index],":comm_idle")) {       /* comm overhead */
    inc_index(1);
    if (is_not_default)
      cmd.comm_idle = atoi (argv[index]);
    inc_index(0);
  } /* :comm_idle */

  else if (streql (argv[index],":first")) {   /* starting search */
    inc_index(1);
    if (is_not_default)
      cmd.first = atoi (argv[index]);
    inc_index(0);
  } /* :first */

  else if (streql (argv[index],":param")) {   /* algorithm parameters */
    inc_index(1);
    if (is_not_default) {
      cmd.nparam = atoi (argv[index]);
      cmd.param = (char **) malloc(cmd.nparam * sizeof(char *));
    }
```

```
      for (i = 0; i < cmd.nparam; i++) {
        inc_index(1);
        *(cmd.param + i) = argv[index];
      }
      inc_index(0);
    } /* :param */

    else error("no such option in cmd line\n");
  } /* while loop */

  if (cmd.accu != 1.0) cmd.approx = accu2approx (cmd.accu);

#ifdef DEBUG
if (cmd.debug >= 1) {
  printf("summary of command and search info:\n");
  printf("counting system=%s\n", dbg_count[toint(cmd.count)]);
  printf("task is %s\n", dbg_task[toint(problem.task)]);
  printf("search strategy=%s\n", dbg_strategy[toint(cmd.strategy)]);
  printf("search algorithm=%s\n", dbg_algorithm[toint(cmd.algorithm)]);
  printf("nsamples=%d\n", cmd.iter);
  printf("problem size=%d\n", problem.size);
  printf("random seed=%d\n", problem.seed);
  printf("stat file=%s\n", io.stat);
  printf("summary file=%s\n", io.summary);
  printf("graph-ready file=%s\n", io.graph);
  printf("run-time file=%s\n", io.rt_pf);
  printf("space-vs-time file=%s\n", io.st_pf);
  printf("report file=%s\n", io.report);
  printf("approx=%g\n", cmd.approx);
  printf("accu=%g\n", cmd.accu);
  printf("debugging mode=%d\n", cmd.debug);
  printf("time constr=");
  if (cmd.constr.time == huge_long) printf("unlimited\n");
  else printf("%d\n", cmd.constr.time);
  printf("space constr=");
  if (cmd.constr.space == huge_long) printf("unlimited\n");
  else printf("%d\n", cmd.constr.space);
  printf("cst constr=");
  if (cmd.constr.cst == huge_float) printf("unlimited\n");
  else printf("%g\n", cmd.constr.cst);
  printf("profile run-time=%s\n", (cmd.pf.rt == YES) ? "YES" : "NO");
  if (cmd.pf.rt == YES) printf("profiling step=%f\n", cmd.pf.step);
  printf("profile space-vs-time=%s\n", (cmd.pf.st == YES) ? "YES" : "NO");
}
#endif

  if (! is_prob_spec) error("problem is not specified\n");

  /* specify some special parameters for Chu */
  switch(cmd.algorithm) {
    case sTCA:  /* :param 1 g */
    case sTCGD: /* :param 1 g */
                sscanf(*(cmd.param), "%f", &(cmd.factor.g)); break;
    case pTCA:  /* :param 2 s c */
    case dTCA:  /* :param 2 s c */
    case pTCGD: /* :param 2 s c */
                sscanf(*(cmd.param), "%f", &(cmd.factor.s));
                sscanf(*(cmd.param+1), "%f", &(cmd.factor.c)); break;
    default:    break;
  }

#ifdef DEBUG
if (cmd.debug >= 1) {
  printf("g_factor=%g\n", cmd.factor.g);
  printf("s_factor=%g\n", cmd.factor.s);
  printf("c_factor=%g\n", cmd.factor.c);
  printf("num_parameters=%d\n", cmd.nparam);
  for (i = 0; i < cmd.nparam; i++) printf("param[%d]=%s\n", i, *(cmd.param+i));
  printf("exit cmd_line processor\n");
}
#endif
}

::::::::::::::
interface/output.c
::::::::::::::

void flush (sp)
  search_ *sp;
{
#ifdef OUT_GRAPH
  flush_graph(sp);
#endif
#ifdef OUT_SUMMARY
  flush_summary(sp);
#endif
}


void flush_summary (sp)
  search_ *sp;
{ FILE *fp;

  fp = fopen(io.summary, "w");
  fprintf(fp, "counting system=%s\n", (cmd.count == REAL) ? "real" : "virtual");

  if (sp->constr.time == huge_long) fprintf(fp, "time unlimited\n");
  else fprintf(fp, "time limit=%d\n", sp->constr.time);

  if (sp->constr.space == huge_long) fprintf(fp, "space unlimited\n");
  else fprintf(fp, "space limit=%d\n", sp->constr.space);

  if (sp->constr.cst == huge_float) fprintf(fp, "cst unlimited\n");
  else fprintf(fp, "cst limit=%g\n", sp->constr.cst);

  fprintf(fp, "real time=%d\n", sp->stat.time);
  fprintf(fp, "real max space=%g\n",
             ((float) (node_conf.node_size * sp->stat.max_active)));
  fprintf(fp, "real cst=%g\n", sp->stat.r_cst);

#ifdef TIME_IS_GEN
  fprintf(fp, "virtual time=%d\n", sp->stat.generated);
#else
  fprintf(fp, "virtual time=%d\n", sp->stat.expanded);
#endif
  fprintf(fp, "virtual max space=%d\n", sp->stat.max_active);
  fprintf(fp, "virtual cst=%g\n", sp->stat.v_cst);

  fprintf(fp, "root approx=%g\n", sp->approx.root_approx);
  fprintf(fp, "run-time approx=%g\n", sp->approx.run_time);

  if ((sp->approx.approx > 0.0) || (sp->approx.run_time > 0.0))
    fprintf(fp,"semi-");
  fprintf(fp, "optimal solution value=%g\n", tofloat(sp->glub));

  fprintf(fp, "cmd-line (or adapted) approx=%g\n", sp->approx.approx);
  fprintf(fp, "achieved approx=%g\n", sp->approx.achieved);
  fprintf(fp, "run-time approx=%g\n", sp->approx.run_time);
```

```
  fclose(fp);
}


void flush_graph (sp)
  search_ *sp;
{ FILE *fp;

  fp = fopen(io.graph,"a");

  if (OUT_TIME_LIMIT) fprintf(fp, "%d  ", sp->constr.time);
  if (OUT_SPACE_LIMIT) fprintf(fp, "%g  ", sp->constr.space);
  if (OUT_CST_LIMIT) fprintf(fp, "%g  ", sp->constr.cst);

  if (OUT_REAL_TIME) fprintf(fp, "%d  ", sp->stat.time);
  if (OUT_REAL_MAX_SPACE) fprintf(fp, "%g  ",
                          ((float) (node_conf.node_size * sp->stat.max_active)));
  if (OUT_REAL_CST) fprintf(fp, "%g  ", sp->stat.r_cst);

  if (OUT_VIRTUAL_TIME)
#ifdef TIME_IS_GEN
    if (cmd.num_pe <= 1) fprintf(fp, "%d  ", sp->stat.generated);
#else
    if (cmd.num_pe <= 1) fprintf(fp, "%d  ", sp->stat.expanded);
#endif
    else fprintf(fp, "%d  ", sp->stat.num_op);
  if (OUT_VIRTUAL_MAX_SPACE) fprintf(fp, "%d  ", sp->stat.max_active);
  if (OUT_VIRTUAL_CST) fprintf(fp, "%g  ", sp->stat.v_cst);

  if (OUT_ROOT_APPROX) fprintf(fp, "%g  ", sp->approx.root_approx);
  if (OUT_RUN_TIME_APPROX) fprintf(fp, "%g  ", sp->approx.run_time);
  if (OUT_APPROX) fprintf(fp, "%g  ", sp->approx.achieved);
  if (OUT_INCUMBENT) fprintf(fp, "%g  ", tofloat(sp->glub));
  if (OUT_LOWB) fprintf(fp, "%g  ", tofloat(sp->gllb));
  if (OUT_THRESHOLD) fprintf(fp, "%g  ", tofloat(sp->constr.bound));

  switch (sp->algorithm) {
    case sTCA:
      if (OUT_GRADIENT_FACTOR) fprintf(fp, "%g  ", sp->alg.stca->g_factor);
      break;
    case sTCGD:
      if (OUT_GRADIENT_FACTOR) fprintf(fp, "%g  ", sp->alg.stcgd->g_factor);
      break;
    case pTCA:
      if (OUT_STOPPING_FACTOR) fprintf(fp, "%g  ", sp->alg.ptca->s_factor);
      if (OUT_CORRECTIVE_FACTOR) fprintf(fp, "%g  ", sp->alg.ptca->c_factor);
      break;
    case dTCA:
      if (OUT_STOPPING_FACTOR) fprintf(fp, "%g  ", sp->alg.dtca->s_factor);
      if (OUT_CORRECTIVE_FACTOR) fprintf(fp, "%g  ", sp->alg.dtca->c_factor);
      break;
    case pTCGD:
      if (OUT_STOPPING_FACTOR) fprintf(fp, "%g  ", sp->alg.ptcgd->s_factor);
      if (OUT_CORRECTIVE_FACTOR) fprintf(fp, "%g  ", sp->alg.ptcgd->c_factor);
      break;
    default: break;
  }
  fprintf(fp, "\n");  fclose(fp);
}

::::::::::::::
interface/result.c
::::::::::::::
```

```
void keep_result (iter, sp)
  int iter;
  search_ *sp;
{ keep_stat[iter] = sp->stat; }


void print_result (iter)
  int iter;
{ int i;
  long  gen = 0, expnd = 0;
  float cst = 0.0, fiter = tofloat(iter);
  FILE  *fp;

  for (i = 1; i <= iter; i++) {
    gen += keep_stat[i].generated;
    expnd += keep_stat[i].expanded;
    cst += keep_stat[i].v_cst;
  }

  fp = fopen(io.stat, "w");
  fprintf(fp, "(");
  fprintf(fp, "(%f %f %f) ", ((float) gen) / fiter,
                             ((float) expnd) / fiter,
                             cst / fiter);
  for (i = 1; i <= iter; i++)
    fprintf(fp, " (%d %d %g) ", keep_stat[i].generated,
                                keep_stat[i].expanded,
                                keep_stat[i].v_cst);
  fprintf(fp, ")\n");
  fclose(fp);
}
```

```
Thu Jan 30 15:30:49 CST 1992
::::::::::::::
algorithm/default.c
::::::::::::::

void default_algorithm (sp)
  search_ *sp;
{ search_message_ msg;

  set_search_message(&msg, FRESH_START, NULL, NO, cmd.pf.rt, cmd.pf.st);
  sp->approx.achieved = sp->approx.root_approx;
  switch (sp->strategy) {
    case GBB:  gbb_primitive(sp, NULL, &msg);           break;
    case BFS:  bfs_primitive(sp, NULL, &msg);           break;
    case DFS:  dfs_primitive(sp, NULL, &msg);           break;
    case GDFS: gdfs_primitive(sp, NULL, &msg);          break;
    case BAND: band_primitive (sp, NULL, &msg);         break;
    default:   error("search: no such strategy\n");     break;
  }
  eval_final_approx(sp);
}

::::::::::::::
algorithm/ida.c
::::::::::::::

/*+ ------------------------------------------------- +
  +              IDA* Search Algorithms               +
  + ------------------------------------------------- +*/

#define PRINT_ITER_PROFILE      printf \
("iter=%d, approx=%g, accu=%g, exp_opt(glub=%g)=%g, threshold=%g, time=%d\n", \
iter, child_sp->approx.approx, approx2accu (child_sp->approx.approx), \
(float) child_sp->glub, (float) expected_opt(child_sp), \
(float) child_sp->constr.bound, get_time(child_sp));

void set_ida_device ();


void ida_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  search_ending_ term_signal; /** not important, only as a placeholder **/
  int iter = 0;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;

  set_search_message (&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  child_sp = create_child_search (sp);

  while (++iter) {
    set_ida_device (child_sp, iter);
    put_child_constr (child_sp, sp, 1.0, 1.0, 1.0);

    idpd_init();
    msg.feasible = NO;
    switch (sp->strategy) {
    case DFS:  term_signal = dfs_primitive (child_sp, sp, &msg); break;
    case GDFS: term_signal = gdfs_primitive (child_sp, sp, &msg); break;
    case BAND: term_signal = band_primitive (child_sp, sp, &msg); break;
    default:   error ("ida_algorithm: no such feasible primitive"); break;
    }

    eval_final_approx (child_sp);
#ifdef DEBUG
if (cmd.debug >= 1) { PRINT_ITER_PROFILE }
#endif
    merge_stat_to_parent (child_sp, sp);
    merge_sol_to_parent (child_sp, sp);
    if (get_approx (child_sp) <= sp->approx.achieved)
      merge_approx_to_parent (child_sp, sp);
    merge_threshold_to_parent (child_sp, sp);

    if (msg.feasible == YES || get_time (sp) >= sp->constr.time) break;

     (sp->child = child_sp = create_brother_search (child_sp))->parent = sp;
  }
}


void set_ida_device (sp, iter)
  search_ *sp;
  int iter;
{
  if (iter == 1) sp->constr.bound = ROOT->lowb+1;
  else
    if (sp->algorithm == DFS_star) sp->constr.bound *= 2;
    else {
      sp->constr.bound = sp->brother->constr.bound + 1;
      if (sp->constr.bound < Next_IDA_Threshold)
        sp->constr.bound = Next_IDA_Threshold;
    }

  Next_IDA_Threshold = problem.huge;
}

::::::::::::::
algorithm/ira.c
::::::::::::::

/*              IRA*                  */

void ira_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  search_ending_ term_signal;
  int iter;
  float a[4];

  a[0] = 0.2; a[1] = 0.1; a[2] = 0.05; a[3] = 0.0;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;

  set_search_message (&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  child_sp = create_child_search (sp);
  for (iter = 0; iter < 4; iter++) {
    inherit_sol_from_parent (child_sp, sp);
    child_sp->approx.approx = a[iter];
    idpd_init ();
    term_signal = (sp->strategy == DFS) ? dfs_primitive(child_sp, sp, &msg) :
                                          gdfs_primitive(child_sp, sp, &msg);
    eval_final_approx(child_sp);
```

```
#ifdef DEBUG
if (cmd.debug >= 1)
  printf("iter=%d, approx(glub=%g)=%g, threshold=%g, time=%d\n",
        iter, (float) child_sp->glub, (float) expected_opt(child_sp),
        (float) child_sp->alg.urts->th, get_time(child_sp));
#endif
    merge_stat_to_parent (child_sp, sp);
    merge_sol_to_parent (child_sp, sp);
    if (get_approx (child_sp) <= sp->approx.achieved) {
      merge_approx_to_parent (child_sp, sp);
    }
    (sp->child = child_sp = create_brother_search (child_sp))->parent = sp;
  }
}
:::::::::::::
algorithm/ird.c
:::::::::::::
/*+ ---------------------------------------------------- +
  +            IRD* Search Algorithms                    +
  + algorithms:                                          +
  +   . unary IRD* (uIRD*): u[a] & u[th]                 +
  +     using only one pruning device.                   +
  +   . binary IRD* (bIRD*):                             +
  +     using both pruning devices.                      +
  + ---------------------------------------------------- +*/


void set_uird_device (), set_uird_approx (), set_uird_threshold ();
void set_bird_device (), set_bird_approx (), set_bird_threshold ();
void ird_set_typeI_approx (), ird_set_typeII_approx ();
void ird_set_regress_approx (), ird_set_last_approx ();
void ird_set_ird_threshold (), ird_set_naive_threshold ();
void ird_set_static_threshold (), ird_set_typeI_static_threshold ();
void ird_set_typeII_static_threshold (), ird_set_predictive_threshold ();
void ird_set_typeI_predictive_threshold ();
void ird_set_typeII_predictive_threshold ();
void reset_th_environ (), set_th_entry ();

float get_g_factor (), get_r_factor ();
domain ird_get_predicted_glub ();
yesno_ check_span (), get_bin_vs_time (), get_th_vs_time ();
yesno_ get_glub_vs_time (), get_approx_vs_time (), get_glub_vs_approx ();
yesno_ get_accu_vs_time ();
iter_attr_ get_iteration_attribute ();
set_th_ *get_set_th ();


/*              uIRD*                   */

void uird_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  search_ending_ term_signal;
  yesno_ break_signal = NO;
  int iter;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;
  set_search_message(&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  child_sp = create_child_search(sp);
  break_signal = NO;
  for (iter = 1; 1; iter++) {
```

```
    RTS_IRD_iter = iter;
#ifndef NO_INHERIT
    inherit_sol_from_parent(child_sp, sp);
#endif
    put_child_constr(child_sp, sp, 1.0, 1.0, 1.0);
    set_uird_device(child_sp, iter);
    idpd_init();
    term_signal = (sp->strategy == DFS) ? dfs_primitive(child_sp, sp, &msg) :
                                          gdfs_primitive(child_sp, sp, &msg);
    eval_final_approx(child_sp);
#ifdef DEBUG
if (cmd.debug >= 1)
  printf("iter=%d, approx(glub=%g)=%g, threshold=%g, time=%d\n",
        iter, (float) child_sp->glub, (float) expected_opt(child_sp),
        (float) child_sp->alg.uird->th, get_time(child_sp));
#endif
    merge_stat_to_parent(child_sp, sp);
    if (get_approx(child_sp) <= sp->approx.achieved) {
      merge_approx_to_parent(child_sp, sp);
      merge_sol_to_parent(child_sp, sp);
    }
    merge_threshold_to_parent(child_sp, sp);

    switch (sp->alg.uird->alg) {
      case UNARY_APPROX:
        if (child_sp->approx.approx <= 0.0) break_signal = YES;
        break;
      case UNARY_TH:
        if (msg.feasible == YES) break_signal = YES;
        break;
      default:  error("uird_algorithm: no such pruning device");  break;
    }
    if (break_signal == YES) break;
    msg.feasible = NO;

    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
  }
}


void set_uird_device (sp, iter)
  search_ *sp;
  int iter;
{
  switch (sp->alg.uird->alg) {
    case UNARY_APPROX:  set_uird_approx(sp, iter);     break;
    case UNARY_TH:      set_uird_threshold(sp, iter);  break;       \
    default:     error("set_uird_device: no such pruning device");  break;
  }
}


void set_uird_approx (sp, iter)
  search_ *sp;
  int iter;
{
  switch (sp->alg.uird->a_type) {
    case TYPE_I:  ird_set_typeI_approx(sp, iter, &(sp->alg.uird->basic)); break;
    case TYPE_II: ird_set_typeII_approx(sp, iter, &(sp->alg.uird->basic)); break;
    default: error("set_uird_approx: no such type"); break;
  }
}
```

```
void set_uird_threshold (sp, iter)
  search_ *sp;
  int iter;
{ uird_ *alg = sp->alg.uird;
  domain ub = sp->parent->root_ub;
  domain lb = sp->parent->root_lb;
  float g = get_g_factor (sp);

  if (iter == 1) {
    /* set init uird threshold */
    sp->constr.bound = alg->th = lb + g * (ub - lb);
  }
  else { /* set intermediate-iteration uird threshold */
    ird_set_ird_threshold(sp, iter, &(alg->basic));
  }

  reset_th_environ(sp);
}


void uird_entry (sp, node)
  search_ *sp;
  node_ *node;
{
  switch (sp->alg.uird->alg) {
    case UNARY_APPROX:                             break;
    case UNARY_TH:      set_th_entry(sp, node); break;
    default: error("uird_entry: no such pruning device");  break;
  }
}


/*               bIRD*                     */
void bird_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  search_ending_ term_signal;
  int iter;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;
  set_search_message(&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  child_sp = create_child_search(sp);
  for (iter = 1; 1; iter++) {
    RTS_IRD_iter = iter;
#ifndef NO_INHERIT
    inherit_sol_from_parent(child_sp, sp);
#endif
    put_child_constr(child_sp, sp, 1.0, 1.0, 1.0);
    set_bird_device(child_sp, iter);
    idpd_init();
    term_signal = (sp->strategy == DFS) ? dfs_primitive(child_sp, sp, &msg) :
                                          gdfs_primitive(child_sp, sp, &msg);
    eval_final_approx(child_sp);
#ifdef DEBUG
if (cmd.debug >= 1)
  printf("iter=%d, approx(glub=%g)=%g, threshold=%g, time=%d\n",
        iter, (float) child_sp->glub, (float) expected_opt(child_sp),
        (float) child_sp->alg.bird->th, get_time(child_sp));
#endif
    merge_stat_to_parent(child_sp, sp);
    if (get_approx(child_sp) <= sp->approx.achieved) {
      merge_approx_to_parent(child_sp, sp);
      merge_sol_to_parent(child_sp, sp);
    }
    merge_threshold_to_parent(child_sp, sp);
    if (child_sp->constr.bound == problem.huge) {
      if (child_sp->approx.approx <= 0.0) break;
    }
    else if (msg.feasible == YES) break;
    msg.feasible = NO;
    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
  }
}


void set_bird_device (sp, iter)
  search_ *sp;
  int iter;
{
    /* Ideally, approx and threshold should be set iteratively,
     * until both converge.
     * However, it may not converge and here it is emulated by
     * setting approx first and then setting threshold.
     */
    set_bird_approx(sp, iter);
    set_bird_threshold(sp, iter);
}


void set_bird_approx (sp, iter)
  search_ *sp;
  int iter;
{
  ird_set_typeI_approx(sp, iter, &(sp->alg.bird->basic));
}


void set_bird_threshold (sp, iter)
  search_ *sp;
  int iter;
{ bird_ *alg = sp->alg.bird;
  domain ub = sp->parent->root_ub;
  domain lb = sp->parent->root_lb;
  float g = get_g_factor (sp);

  if (iter == 1) {
    /* set init bird threshold */
    sp->constr.bound = alg->th = lb + g * (ub - lb);
  }
  else { /* set intermediate-iteration uird threshold */
    ird_set_ird_threshold(sp, iter, &(alg->basic));
  }

  reset_th_environ(sp);
}


void bird_entry (sp, node)
  search_ *sp;
  node_ *node;
{ set_th_entry(sp, node); }


/*+ ------------------------------------------------- +
  +       IRD* Approximation Setting Routines          +
  + algorithms:                                          +
  +   . naive approximation (Type I):                    +
  +     linear gradient, like sTCGD.                     +
  +   . regression approximation (Type II):              +
  +     regressed gradient, like pTCGD.                  +
  +   * last approximation: regression gradient.         +
  + --------------------------------------------------- +*/


/** linear-gradient approximation **/
void ird_set_typeI_approx (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ float sched;
  float x_root = sp->parent->approx.x_root_approx;
  float g = get_g_factor (sp);
  int count;
  search_ *brother;
  float glub[2];

  if (cmd.xon == YES) {
    x_root = approx2accu (x_root);
    sched = x_root + (1.0 - x_root) * iter * g;
  } else {
    sched = x_root * (1.0 - iter * g);
  }

  if (basicp->shortcut && check_span (sp, FULL_SPAN) == YES) {

    count = 0;
    for (brother = sp->brother; brother; brother = brother->brother)
      if (is_feasible_time (get_time (brother))) {
        glub[count++] = (float) (brother->glub);
        if (count >= 2) break;
      } else break;

    if (glub[1] - glub[0] <= glub[0] * basicp->margin)
      /** if the upper bounds changes only a little bit,
       ** then it is likely that upb is very close to optimum. **/
      sched = (cmd.xon == YES) ? 1.0 : 0.0;
  }

  if (cmd.xon == YES) sched = accu2approx (sched);
  sp->approx.approx = (sched >= 0.0) ? sched : 0.0;
}


void ird_set_regress_approx (sp, iter, span, basicp)
  search_ *sp;
  int iter;
  span_ span;
  ird_basic_ *basicp;
{ float sched, a;
  double beta[2];
  float x_root = sp->parent->approx.x_root_approx;
  float g = get_g_factor (sp);

  if (cmd.xon == YES) {
    x_root = approx2accu (x_root);
    sched = x_root + (1.0 - x_root) * iter * g;
  } else {
    sched = x_root * (1.0 - iter * g);
  }

  if (cmd.xon == YES) {
    if (get_accu_vs_time(sp, beta, span) == YES) {
      sched = beta[0] + beta[1] * log10(((double) sp->constr.time));
      a = get_approx(sp->parent);
      a = approx2accu (a);
      if (sched <= a) sched = a + (1.0 - x_root) * g;
    }
    sched = accu2approx (sched);
  } else {
    if (get_approx_vs_time(sp, beta, span) == YES) {
      sched = beta[0] + beta[1] * log10(((double) sp->constr.time));
      if (sched >= get_approx(sp->parent))
        sched = get_approx(sp->parent) - sp->parent->approx.root_approx * g;
    }
  }

  sp->approx.approx = (sched >= 0.0) ? sched : 0.0;
}


void ird_set_typeII_approx (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ ird_set_regress_approx (sp, iter, FULL_SPAN, basicp); }



/*+ --------------------------------------------------- +
  +           IRD* Threshold Setting Routines           +
  + algorithms:                                         +
  +   . naive algorithm.                                +
  +   . static algorithm: type I & type II              +
  +   . predictive algorithm: type I & type II          +
  + --------------------------------------------------- +*/


void ird_set_ird_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{
  switch ((get_set_th(sp))->alg) {
    case NAIVE_TH:      ird_set_naive_threshold(sp, iter, basicp);      break;
    case STATIC_TH:     ird_set_static_threshold(sp, iter, basicp);     break;
    case PREDICTIVE_TH: ird_set_predictive_threshold(sp, iter, basicp); break;
    default: error("ird_set_ird_threshold: no such threshold setting"); break;
  }
}


void ird_set_naive_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  float g = get_g_factor (sp);
  domain th;
  double beta[2];
  int count;
  search_ *brother;
  float glub[2];
  sp->constr.bound = lb + iter * g * (ub - lb);

  if (basicp->shortcut && check_span (sp, FULL_SPAN) == YES) {

    count = 0;
    for (brother = sp->brother; brother; brother = brother->brother)
      if (is_feasible_time (get_time (brother))) {
        glub[count++] = (float) (brother->glub);
        if (count >= 2) break;
      } else break;

    if (glub[1] - glub[0] <= glub[0] * basicp->margin)
      /** if the upper bounds changes only a little bit,
       ** then it is likely that upb is very close to optimum. **/
      sp->constr.bound = glub[0];
  }

  put_threshold (sp, sp->constr.bound);
}


void ird_set_static_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{
  switch ((get_set_th(sp))->type) {
    case TYPE_I:   ird_set_typeI_static_threshold(sp, iter, basicp);  break;
    case TYPE_II:  ird_set_typeII_static_threshold(sp, iter, basicp); break;
    default:       error("ird_set_static_threshold: no such type");   break;
  }
}


void ird_set_typeI_static_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ domain u = ird_get_predicted_glub (sp, iter, basicp);

  sp->constr.bound = (u + sp->brother->glub) * 0.5 / (1.0 + sp->approx.approx);
  put_threshold(sp, sp->constr.bound);
}


void ird_set_typeII_static_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ search_ *brother = sp->brother;
  domain prev_th = get_threshold (sp->brother);
  domain prev_prev_glub, prev_glub, dth;
  float g = get_g_factor (sp);
  float ub, lb, delta, base;

  if (iter == 2) {
    prev_prev_glub = sp->parent->root_lb;
    prev_glub = brother->glub;
  }
  else {
    prev_prev_glub = brother->brother->glub;
    prev_glub = brother->glub;
  }
  base = 1.0 + brother->approx.approx;
  lb = prev_glub / base;
  ub = prev_prev_glub / base;
  if (prev_th < lb) {
    /* case I */
    delta = (ub - lb) * iter * g;
    dth = lb - prev_th + delta;
  }
  else {
    /* case II */
    dth = delta = (ub - lb) * g;
  }
  sp->constr.bound = prev_th + dth;
  ub = sp->glub / (1.0 + sp->approx.approx);
  if (sp->constr.bound > ub) sp->constr.bound = ub;
  put_threshold(sp, sp->constr.bound);
}


void ird_set_predictive_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{
  switch ((get_set_th(sp))->type) {
    case TYPE_I:   ird_set_typeI_predictive_threshold(sp, iter, basicp);  break;
    case TYPE_II:  ird_set_typeII_predictive_threshold(sp, iter, basicp); break;
    default:       error("ird_set_predictive_threshold: no such type");   break;
  }
}


void ird_set_typeI_predictive_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ search_ *brother = sp->brother;
  domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  float g = get_g_factor(sp);
  long this_time;
  double beta[2];

  sp->constr.bound = lb + iter * g * (ub - lb);

  if (get_th_vs_time (sp, beta, FULL_SPAN) == YES) {
    this_time = get_r_factor (sp) * get_time (brother);
    if (this_time > sp->constr.time) this_time = sp->constr.time;
    sp->constr.bound = beta[0] + beta[1] * log10 ((double) this_time);
    if (sp->constr.bound < brother->constr.bound)
      sp->constr.bound = brother->constr.bound + g * (ub - lb);
  }

  put_threshold (sp, sp->constr.bound);
}


void ird_set_typeII_predictive_threshold (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ search_ *brother = sp->brother;
  domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  float g = get_g_factor(sp);
  long this_time;
  double beta[2];

  sp->constr.bound = lb + iter * g * (ub - lb);

  if (get_bin_vs_time (sp, beta, FULL_SPAN) == YES) {
    this_time = get_r_factor (sp) * get_time (brother);
    if (this_time > sp->constr.time) this_time = sp->constr.time;
    sp->constr.bound = beta[0] + beta[1] * log10 ((double) this_time);
    if (sp->constr.bound < brother->constr.bound)
      sp->constr.bound = brother->constr.bound + g * (ub - lb);
  }

  put_threshold (sp, sp->constr.bound);
}



/*+ -------------------------------------------------- +
  +              IRD*    Library Routines               +
  +          Many Library Routines are in RTS           +
  + -------------------------------------------------- +*/


domain ird_get_predicted_glub (sp, iter, basicp)
  search_ *sp;
  int iter;
  ird_basic_ *basicp;
{ span_ span = FULL_SPAN;
  float step;
  double beta[2];

  if (get_glub_vs_approx (sp, beta, span) == YES)
    return ((domain) (beta[0] + beta[1] * sp->approx.approx));

  /* if situation is too tough to handle, simply use naive glub */
  step = iter * get_g_factor (sp) * (sp->parent->root_ub - sp->parent->root_lb);
  return ((domain) (sp->parent->root_ub - step));
}

::::::::::::::
algorithm/lw.c
::::::::::::::

void lawler_wood_algorithm (sp)
  search_ *sp;
{ constr_ save_constr;
  search_message_ msg;

  sp->approx.achieved = sp->approx.root_approx;
  save_constr = sp->constr;
  set_stop_constr(sp, 0.5, 0.5, 0.5);
  set_search_message(&msg, FRESH_START, NULL, NO, cmd.pf.rt, cmd.pf.st);
  while (bfs_primitive(sp, NULL, &msg) == SEARCH_IS_ABORTED) {
    if (is_lw_end(sp, &save_constr) == YES) break;
    msg.style = RESUME;
    sp->approx.approx += 0.05;
    lw_set_constr(sp, &save_constr);
  }
  sp->constr = save_constr;
  eval_final_approx(sp);
}
```

```
yesno_ is_lw_end (sp, constrp)
  search_ *sp;
  constr_ *constrp;
{
  if (sp->constr.time != huge_long)
    if (sp->constr.time >= constrp->time) return YES;
  if (sp->constr.space != huge_long)
    if (sp->constr.space >= constrp->space) return YES;
  if (sp->constr.cst != huge_float)
    if (sp->constr.cst >= constrp->cst) return YES;
  return NO;
}

void lw_set_constr (sp, constrp)
  search_ *sp;
  constr_ *constrp;
{ float time, space;

  if (sp->constr.time != huge_long) {
    time = (float) (constrp->time - sp->constr.time);
    time = time * 0.5;
    sp->constr.time = tolong(time);
  }
  if (sp->constr.space != huge_long) {
    space = (float) (constrp->space - sp->constr.space);
    space = space * 0.5;
    sp->constr.space = tolong(space);
  }
  if (sp->constr.cst != huge_float)
    sp->constr.cst += ((constrp->cst - sp->constr.cst) * 0.5);
}

::::::::::::::
algorithm/para.c
::::::::::::::
/**
 **              Parallel Processing of Searches
 **/

void para_algorithm (sp, pe, num_pe)
  search_ *sp;
  search_ *pe[];
  int num_pe;
{ search_message_ msg;
  yesno_ break_signal;
  int i;
  search_ *first_sp;
  node_ *rootlet;
  long *ngen_arr = NULL;
  long max_ngen;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;

  /** start the first bfs **/
  first_sp = create_child_search (sp);
  set_search_message (&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  idpd_init ();
  put_child_constr (first_sp, sp, 1.0, 1.0, 1.0);
  switch (cmd.first)
  {
    case -1:
      first_sp->strategy = BFS;
      first_sp->algorithm = DEFAULT;
      first_sp->open.btree = NULL;
      pfirst_primitive (first_sp, sp, &msg, 1);
      break;
    default:
      first_sp->strategy = BFS;
      first_sp->algorithm = DEFAULT;
      first_sp->open.btree = NULL;
      pfirst_primitive (first_sp, sp, &msg, num_pe);
      break;
  }
  merge_sol_to_parent (first_sp, sp);
  merge_stat_to_parent (first_sp, sp);

  /** create an array of ngen **/
  ngen_arr = (long *) malloc (num_pe * sizeof (long));

  /** create a search process for each pe **/
  for (i = 0; i < num_pe; i++)
  {
    pe[i] = create_child_search(sp);
    pe[i]->strategy = cmd.strategy;
    pe[i]->algorithm = cmd.algorithm;
    pe[i]->open.list = NULL;
    pe[i]->incumbent = sp->incumbent;
    pe[i]->glub = sp->glub;
    pe[i]->gllb = sp->gllb;
    if ((rootlet = btree_delete (first_sp))) insert (rootlet, pe[i]);
    put_child_constr (pe[i], sp, 1.0, 1.0, 1.0);
    pe[i]->approx.approx = sp->approx.approx;
  }

  set_search_message (&msg, RESUME, NULL, YES, cmd.pf.rt, cmd.pf.st);

  break_signal = NO;
  while (para_termination (pe, num_pe) == NO)
  {
    /** start parallel search **/
    for (i = 0; i < num_pe; i++)
      break_signal = para_primitive (pe[i], sp, &msg, sp->gllb, ngen_arr+i);

#ifdef TIME_IS_GEN
    max_ngen = 0;
    for (i = 0; i < num_pe; i++)
      if (max_ngen < *(ngen_arr+i)) max_ngen = *(ngen_arr+i);
    sp->stat.num_op += max_ngen;
#else
    /** it is assumed that the virtual time is the number of nodes expanded **/
    (sp->stat.num_op)++;
#endif

    /** bookkeeping **/
    para_merge_solution (sp, pe, num_pe);

    if (break_signal == YES) break;

    /** load_balancing **/
    para_load_balancing (pe, num_pe);
  }
  para_merge_stat (sp, pe, num_pe);
  eval_final_approx(sp);
}
```

```
yesno_ para_primitive (pep, sp, msgp, gllb, ngenp)
  search_ *pep;
  search_ *sp;
  search_message_ *msgp;
  domain gllb;
  long *ngenp;
{ yesno_ break_signal = NO;
  search_ending_ retval;
  long ngen;

  switch (cmd.algorithm) {
  case PBFS:
    if (pbfs_primitive (pep, sp, msgp, gllb, ngenp) == SEARCH_IS_ABORTED)
      break_signal = YES;
    break;
  case PGDFS:
    if (pgdfs_primitive (pep, sp, msgp, gllb, ngenp) == SEARCH_IS_ABORTED)
      break_signal = YES;
    break;
  default:
    error ("para_primitive: no such parallel search primitive");
    break;
  }
  return break_signal;
}
::::::::::::::
algorithm/rts.c
::::::::::::::
/* begin, shorthand */
#define PRINT_ITER_PROFILE      printf \
("iter=%d, approx=%g, accu=%g, exp_opt(glub=%g)=%g, threshold=%g, time=%d\n", \
iter, child_sp->approx.approx, approx2accu (child_sp->approx.approx), \
(float) child_sp->glub, (float) expected_opt(child_sp), \
(float) child_sp->constr.bound, get_time(child_sp));
/* end, shorthand */

/** internal functional predeclarations **/
void set_urts_device (), set_urts_approx (), set_urts_threshold ();
void set_brts_device (), set_brts_approx (), set_brts_threshold ();
void set_hrts_device (), set_hrts_approx ();
void set_lg_approx (), set_gg_approx (), set_fr_approx (), set_qg_approx ();
void set_regress_approx (), set_last_approx (), set_hrts_threshold ();
void set_rts_threshold (), set_naive_threshold (), set_static_threshold ();
void set_lg_threshold (), set_gg_threshold (), set_qg_threshold ();
void set_pr_threshold (), set_mg_threshold ();
void set_predictive_threshold (), set_fr_threshold ();
void set_ld_threshold (), reset_th_environ (), reset_th_bin ();
void set_th_entry (), enter_th_bin ();

float get_g_factor (), get_r_factor ();
domain get_predicted_glub ();
yesno_ check_span (), get_bin_vs_time (), get_th_vs_time ();
yesno_ get_glub_vs_time (), get_approx_vs_time (), get_glub_vs_approx ();
yesno_ get_accu_vs_time (), is_null_device ();
iter_attr_ get_iteration_attribute ();
set_th_ *get_set_th ();



/*              uRTS*                   */

void urts_algorithm (sp)
```

```
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  search_ending_ term_signal;
  float adjust = 1.0;
  float r;
  iter_attr_ attr = OTHER_ITER;
  yesno_ break_signal = NO;
  int iter;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;

  set_search_message(&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  child_sp = create_child_search(sp);
  break_signal = NO;

  r = get_r_factor (sp);
  if (r == 0.0) r = 1.0;

  for (iter = 1; 1; iter++) {
    RTS_IRD_iter = iter;
#ifndef NO_INHERIT
    inherit_sol_from_parent(child_sp, sp);
#endif
    if (child_sp->alg.urts->last_pred == YES) {
      attr = get_iteration_attribute (child_sp, iter);
      switch (attr) {
        case LAST_ITER:     adjust = 1.0;     break;
        case LAST_2ND_ITER: adjust = 1.0 / r; break;
        case OTHER_ITER:    adjust = 1.0;     break;
        default: error("urts_algorithm: no such iteration attribute"); break;
      }
    }

    set_urts_device(child_sp, iter, attr);
    if (is_null_device (child_sp) == YES) adjust = 1.0;
    put_child_constr(child_sp, sp, adjust, 1.0, 1.0);
    idpd_init();
    term_signal = (sp->strategy == DFS) ? dfs_primitive(child_sp, sp, &msg) :
                                          gdfs_primitive(child_sp, sp, &msg);
    eval_final_approx(child_sp);
#ifdef DEBUG
if (cmd.debug >= 1) { PRINT_ITER_PROFILE }
#endif
    merge_stat_to_parent(child_sp, sp);
    merge_sol_to_parent(child_sp, sp);
    if (get_approx(child_sp) <= sp->approx.achieved)
      merge_approx_to_parent(child_sp, sp);
    merge_threshold_to_parent(child_sp, sp);

    switch (sp->alg.urts->alg) {
      case UNARY_APPROX:
        if (child_sp->approx.approx <= 0.0) break_signal = YES;
        break;
      case UNARY_TH:
        if (msg.feasible == YES) break_signal = YES;
        break;
      default:  error("urts_algorithm: no such pruning device");  break;
    }
    if (break_signal == YES) break;
    if (get_time(sp) >= sp->constr.time) break;
    msg.feasible = NO;
```

```
    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
  }
}


void set_urts_device (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch (sp->alg.urts->alg) {
    case UNARY_APPROX:  set_urts_approx(sp, iter, attr);       break;
    case UNARY_TH:      set_urts_threshold(sp, iter, attr);    break;
    default:    error("set_urts_device: no such pruning device");  break;
  }
}


void set_urts_approx (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch (sp->alg.urts->a_type) {
    case TYPE_I:   set_lg_approx(sp, iter);   break;
    case TYPE_II:  set_gg_approx(sp, iter);  break;
    case TYPE_III: set_fr_approx(sp, iter); break;
    case TYPE_IV:  set_qg_approx(sp, iter);  break;
    default: error("set_urts_approx: no such type"); break;
  }
}


void set_urts_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ urts_ *alg = sp->alg.urts;
  domain ub = sp->parent->root_ub;
  domain lb = sp->parent->root_lb;
  float g = get_g_factor (sp);

  if (iter == 1) {
    /* set init urts threshold */
    sp->constr.bound = alg->th = lb + g * (ub - lb);
  }
  else {
    /* set intermediate-iteration urts threshold */
    set_rts_threshold(sp, iter, attr);
  }

  reset_th_environ(sp);
}


void urts_entry (sp, node)
  search_ *sp;
  node_ *node;
{
  switch (sp->alg.urts->alg) {
    case UNARY_APPROX:                            break;
    case UNARY_TH:      set_th_entry(sp, node); break;
    default: error("urts_entry: no such pruning device");  break;
  }
}


/*              bRTS*                    */
void brts_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  search_ending_ term_signal;
  float adjust = 1.0;
  float r;
  iter_attr_ attr = OTHER_ITER;
  int iter;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;

  set_search_message(&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  child_sp = create_child_search(sp);

  r = get_r_factor (sp);
  if (r == 0.0) r = 1.0;

  for (iter = 1; 1; iter++) {
    RTS_IRD_iter = iter;
#ifndef NO_INHERIT
    inherit_sol_from_parent(child_sp, sp);
#endif
    if (child_sp->alg.brts->last_pred == YES)
      switch ((attr = get_iteration_attribute(child_sp, iter))) {
        case LAST_ITER:     adjust = 1.0;     break;
        case LAST_2ND_ITER: adjust = 1.0 / r; break;
        case OTHER_ITER:    adjust = 1.0;     break;
        default: error("brts_algorithm: no such iteration attribute"); break;
      }

    set_brts_device(child_sp, iter, attr);
    if (is_null_device (child_sp) == YES) adjust = 1.0;
    put_child_constr(child_sp, sp, adjust, 1.0, 1.0);
    idpd_init();
    term_signal = (sp->strategy == DFS) ? dfs_primitive(child_sp, sp, &msg) :
                                          gdfs_primitive(child_sp, sp, &msg);
    eval_final_approx(child_sp);
#ifdef DEBUG
if (cmd.debug >= 1) { PRINT_ITER_PROFILE }
#endif
    merge_stat_to_parent(child_sp, sp);
    merge_sol_to_parent(child_sp, sp);
    if (get_approx(child_sp) <= sp->approx.achieved) {
      merge_approx_to_parent(child_sp, sp);
    }
    merge_threshold_to_parent(child_sp, sp);
    if (child_sp->constr.bound == problem.huge) {
      if (child_sp->approx.approx <= 0.0) break;
    }
    else if (msg.feasible == YES) break;
    if (get_time(sp) >= sp->constr.time) break;
    msg.feasible = NO;
    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
  }
}
```

```
void set_brts_device (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
    /* Ideally, approx and threshold should be set iteratively,
     * until both converge.
     * However, it may not converge and here it is emulated by
     * setting approx first and then setting threshold.
     */
    set_brts_approx(sp, iter, attr);
    set_brts_threshold(sp, iter, attr);
}


void set_brts_approx (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch (sp->alg.brts->a_type) {
    case TYPE_I:   set_lg_approx(sp, iter);   break;
    case TYPE_II:  set_gg_approx(sp, iter);  break;
    case TYPE_III: set_fr_approx(sp, iter); break;      /** NA **/
    case TYPE_IV:  set_qg_approx(sp, iter);  break;
    default: error("set_brts_approx: no such type"); break;
  }
}


void set_brts_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ brts_ *alg = sp->alg.brts;
  domain ub = sp->parent->root_ub;
  domain lb = sp->parent->root_lb;
  float g = get_g_factor (sp);

  if (iter == 1) {
    /* set init brts threshold */
    sp->constr.bound = alg->th = lb + g * (ub - lb);
  }
  else {
    /* set intermediate-iteration urts threshold */
    set_rts_threshold(sp, iter, attr);
  }

  /** if approximation degree is 0, then there is no thresholding **/
  if (sp->approx.approx == 0.0) {
    sp->constr.bound = problem.huge;
    put_threshold (sp, sp->constr.bound);
  }

  reset_th_environ(sp);
}


void brts_entry (sp, node)
  search_ *sp;
  node_ *node;
{ set_th_entry(sp, node); }
```

```
/*                hRTS*                    */

void hrts_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  search_ending_ term_signal;
  float adjust = 1.0;
  float r;
  iter_attr_ attr = OTHER_ITER;
  int iter;

  sp->approx.achieved = sp->approx.root_approx;
  sp->gllb = sp->root_lb;
  sp->glub = sp->root_ub;

  set_search_message(&msg, FRESH_START, NULL, YES, cmd.pf.rt, cmd.pf.st);
  child_sp = create_child_search(sp);

  r = get_r_factor (sp);
  if (r == 0.0) r = 1.0;

  for (iter = 1; 1; iter++) {
    RTS_IRD_iter = iter;
#ifndef NO_INHERIT
    inherit_sol_from_parent(child_sp, sp);
#endif
    if (child_sp->alg.hrts->last_pred == YES)
      switch ((attr = get_iteration_attribute(child_sp, iter))) {
        case LAST_ITER:     adjust = 1.0;     break;
        case LAST_2ND_ITER: adjust = 1.0 / r; break;
        case OTHER_ITER:    adjust = 1.0;     break;
        default: error("hrts_algorithm: no such iteration attribute"); break;
      }

    set_hrts_device(child_sp, iter, attr);
    if (is_null_device (child_sp) == YES) adjust = 1.0;
    put_child_constr(child_sp, sp, adjust, 1.0, 1.0);
    idpd_init();
    term_signal = (sp->strategy == DFS) ? dfs_primitive(child_sp, sp, &msg) :
                                         gdfs_primitive(child_sp, sp, &msg);
    eval_final_approx(child_sp);
#ifdef DEBUG
if (cmd.debug >= 1) { PRINT_ITER_PROFILE }
#endif
    merge_stat_to_parent(child_sp, sp);
    merge_sol_to_parent(child_sp, sp);
    if (get_approx(child_sp) <= sp->approx.achieved)
      merge_approx_to_parent(child_sp, sp);
    merge_threshold_to_parent(child_sp, sp);
    if (child_sp->constr.bound == problem.huge) {
      if (child_sp->approx.approx <= 0.0) break;
    } else {
      if (msg.feasible == YES) break;
    }
    if (get_time(sp) >= sp->constr.time) break;
    msg.feasible = NO;
    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
  }
}


void set_hrts_device (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  /* Only the search of last iteration needs both approx and threshold,
   * all searches of other iterations needs approx only.
   */
  set_hrts_approx(sp, iter, attr);

  if (attr == LAST_ITER) {
    set_hrts_threshold(sp, iter, attr);

    /** if approximation degree is 0, then there is no thresholding **/
    if (sp->approx.approx == 0.0) {
      sp->constr.bound = problem.huge;
      put_threshold (sp, sp->constr.bound);
    }
  }
}


void set_hrts_approx (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch (sp->alg.hrts->a_type) {
    case TYPE_I:   set_lg_approx(sp, iter);    break;
    case TYPE_II:  set_gg_approx(sp, iter);   break;
    case TYPE_III: set_fr_approx(sp, iter);  break;
    case TYPE_IV:  set_qg_approx(sp, iter);  break;
    default: error("set_hrts_approx: no such type"); break;
  }
}


void set_hrts_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ hrts_ *alg = sp->alg.hrts;
  domain ub = sp->parent->root_ub;
  domain lb = sp->parent->root_lb;
  float g = get_g_factor (sp);

  if (iter == 1) {
    /* set init hrts threshold **/
    sp->constr.bound = alg->th = lb + g * (ub - lb);
  } else {
    set_rts_threshold(sp, iter, attr);
  }

  reset_th_environ(sp);
}


void hrts_entry (sp, node)
  search_ *sp;
  node_ *node;
{ set_th_entry(sp, node); }


/*+ ------------------------------------------------- +
 +        RTS* Approximation Setting Routines              +
 + algorithms:                                             +
 +   . naive approximation (Type I):                       +
 +     linear gradient, like sTCGD.                        +
 +   . regression approximation (Type II):                 +
 +     regressed gradient, like pTCGD.                     +
 +   * last approximation: regression gradient.            +
 + ---------------------------------------------------- +*/


/** LG: linear-gradient approximation **/
void set_lg_approx (sp, iter)
  search_ *sp;
  int iter;
{ float sched, margin_scale;
  float x_root = sp->parent->approx.x_root_approx;
  float g = get_g_factor (sp);

  if (cmd.xon == YES) {
    x_root = approx2accu (x_root);
    sched = x_root + (1.0 - x_root) * iter * g;
  } else {
    sched = x_root * (1.0 - iter * g);
  }
  if (cmd.xon == YES) sched = accu2approx (sched);
  sp->approx.approx = (sched >= 0.0) ? sched : 0.0;
}


/** GG: geometric-gradient approximation **/
void set_gg_approx (sp, iter)
  search_ *sp;
  int iter;
{ float sched, margin_scale;
  float x_root = sp->parent->approx.x_root_approx;
  float g = get_g_factor (sp);
  int factor = 1;

  if (iter > 2) {
    for (--iter; iter > 0; --iter) factor *= 2;
    iter = factor;
  }

  if (cmd.xon == YES) {
    x_root = approx2accu (x_root);
    sched = x_root + (1.0 - x_root) * iter * g;
  } else {
    sched = x_root * (1.0 - iter * g);
  }
  if (cmd.xon == YES) sched = accu2approx (sched);
  sp->approx.approx = (sched >= 0.0) ? sched : 0.0;
}


/** FR: first-order-regression approximation **/
void set_regress_approx (sp, iter, span)
  search_ *sp;
  int iter;
  span_ span;
{ float sched, a;
  search_ *brother = sp->brother;
  double beta[2];
  float x_root = sp->parent->approx.x_root_approx;
  float g = get_g_factor (sp);
```

```
  double t = (double) sp->constr.time;

  if (cmd.xon == YES) {
    x_root = approx2accu (x_root);
    if (brother)
      sched = approx2accu (brother->approx.approx) + (1.0 - x_root) * g;
    else
      sched = x_root + (1.0 - x_root) * iter * g;
  } else {
    if (brother)
      sched = brother->approx.approx - x_root * g;
    else
      sched = x_root * (1.0 - iter * g);
  }

  if (cmd.xon == YES) {
    if (get_accu_vs_time(sp, beta, span) == YES) {
      sched = beta[0] + beta[1] * log10 (t);
      a = get_approx (sp->parent);
      a = approx2accu (a);
      if (sched <= a) sched = a + (1.0 - x_root) * g;
    }
    sched = accu2approx (sched);
  } else {
    if (get_approx_vs_time(sp, beta, span) == YES) {
      sched = beta[0] + beta[1] * log10 (t);
      a = get_approx (sp->parent);
      if (sched >= a) sched = a - x_root * g;
    }
  }

  sp->approx.approx = (sched >= 0.0) ? sched : 0.0;
}


void set_fr_approx (sp, iter)
  search_ *sp;
  int iter;
{ set_regress_approx (sp, iter, FULL_SPAN); }


void set_last_approx (sp, iter)
  search_ *sp;
  int iter;
{ set_regress_approx (sp, iter, PARTIAL_SPAN); }


/** QG: quadratic-gradient approximation **/
void set_qg_approx (sp, iter)
  search_ *sp;
  int iter;
{ float sched, margin_scale;
  float x_root = sp->parent->approx.x_root_approx;
  float g = get_g_factor (sp);

  iter *= iter;

  if (cmd.xon == YES) {
    x_root = approx2accu (x_root);
    sched = x_root + (1.0 - x_root) * iter * g;
  } else {
    sched = x_root * (1.0 - iter * g);
  }
  if (cmd.xon == YES) sched = accu2approx (sched);
  sp->approx.approx = (sched >= 0.0) ? sched : 0.0;
}


/*+ -------------------------------------------------- +
  +          RTS* Threshold Setting Routines           +
  + algorithms:                                        +
  +   . naive algorithm.                               +
  +   . static algorithm: type I & type II             +
  +   . predictive algorithm: type I & type II         +
  + -------------------------------------------------- +*/


void set_rts_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch ((get_set_th(sp))->alg) {
    case NAIVE_TH:      set_naive_threshold(sp, iter, attr);           break;
    case STATIC_TH:     set_static_threshold(sp, iter, attr);          break;
    case PREDICTIVE_TH: set_predictive_threshold(sp, iter, attr);      break;
    default: error("set_rts_threshold: no such threshold setting");    break;
  }
}

void set_naive_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch ((get_set_th(sp))->type) {
    case TYPE_I:   set_lg_threshold (sp, iter, attr);  break;
    case TYPE_II:  set_gg_threshold (sp, iter, attr); break;
    case TYPE_III: set_gg_threshold (sp, iter, attr);  break;
    default:       error("set_naive_threshold: no such type");  break;
  }
}


/** LG: linear-gradient heuristic **/
void set_lg_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
{ domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  float g = get_g_factor (sp);
  long time = sp->constr.time;
  search_ *brother;
  domain th;
  float scale, glub[2];
  int count;
  double beta[2];

  /** if everything is too tough, simply use it **/
  sp->constr.bound = lb + iter * g * (ub - lb);

  /** if it is the last one, then use prediction **/
  if (attr == LAST_ITER) {
    if (get_th_vs_time (sp, beta, PARTIAL_SPAN) == YES) {
      th = beta[0] + beta[1] * log10 ((double) time);
      if (th < sp->constr.bound) sp->constr.bound = th;
    }
  }

  /** check whether there is no change in upper bound,
   ** if there is no change, then it implies that good suboptimum is found. **/
  if (cmd.nparam >= 7) {
    sscanf(*(cmd.param+6), "%f", &scale);
    glub[0] = glub[1] = huge_float;
    count = 0;

    for (brother = sp->brother; brother; brother = brother->brother)
      if (is_feasible_time (get_time (brother))) {
        glub[count++] = (float) (brother->glub);
        if (count >= 2) break;
      }

    if (glub[1] != huge_float)
      if (glub[1] - glub[0] < glub[0] * scale)
        /** if the upper bounds changes only a little bit,
         ** then it is likely that upb is very close to optimum. **/
        sp->constr.bound = glub[0];
  }

  put_threshold (sp, sp->constr.bound);
}


/** GG: geometric-gradient heuristic **/
void set_gg_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
{ domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  float g = get_g_factor (sp);
  long time = sp->constr.time;
  search_ *brother;
  domain th;
  float scale, glub[2];
  int count;
  double beta[2];
  int factor = 1;

  if (iter > 2) {
    for (--iter; iter > 0; --iter) factor *= 2;
    iter = factor;
  }

  /** if everything is too tough, simply use it **/
  sp->constr.bound = lb + iter * g * (ub - lb);

  /** if it is the last one, then use prediction **/
  if (attr == LAST_ITER) {
    if (get_th_vs_time (sp, beta, PARTIAL_SPAN) == YES) {
      th = beta[0] + beta[1] * log10 ((double) time);
      if (th < sp->constr.bound) sp->constr.bound = th;
    }
  }

  /** check whether there is no change in upper bound,
   ** if there is no change, then it implies that good suboptimum is found. **/
  if (cmd.nparam >= 7) {
    sscanf(*(cmd.param+6), "%f", &scale);
    glub[0] = glub[1] = huge_float;
    count = 0;

    for (brother = sp->brother; brother; brother = brother->brother)
```

```
      if (is_feasible_time (get_time (brother))) {
        glub[count++] = (float) (brother->glub);
        if (count >= 2) break;
      }

    if (glub[1] != huge_float)
      if (glub[1] - glub[0] < glub[0] * scale)
        /** if the upper bounds changes only a little bit,
         ** then it is likely that upb is very close to optimum. **/
        sp->constr.bound = glub[0];
  }

  put_threshold (sp, sp->constr.bound);
}


/** QG: quadratic-gradient heuristic **/
void set_qg_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
{ domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  float g = get_g_factor (sp);
  long time = sp->constr.time;
  search_ *brother;
  domain th;
  float scale, glub[2];
  int count;
  double beta[2];

  iter *= iter;

  /** if everything is too tough, simply use it **/
  sp->constr.bound = lb + iter * g * (ub - lb);

  /** if it is the last one, then use prediction **/
  if (attr == LAST_ITER) {
    if (get_th_vs_time (sp, beta, PARTIAL_SPAN) == YES) {
      th = beta[0] + beta[1] * log10 ((double) time);
      if (th < sp->constr.bound) sp->constr.bound = th;
    }
  }

  /** check whether there is no change in upper bound,
   ** if there is no change, then it implies that good suboptimum is found. **/
  if (cmd.nparam >= 7) {
    sscanf(*(cmd.param+6), "%f", &scale);
    glub[0] = glub[1] = huge_float;
    count = 0;

    for (brother = sp->brother; brother; brother = brother->brother)
      if (is_feasible_time (get_time (brother))) {
        glub[count++] = (float) (brother->glub);
        if (count >= 2) break;
      }

    if (glub[1] != huge_float)
      if (glub[1] - glub[0] < glub[0] * scale)
        /** if the upper bounds changes only a little bit,
         ** then it is likely that upb is very close to optimum. **/
        sp->constr.bound = glub[0];
  }

  put_threshold (sp, sp->constr.bound);
```

```
}


/** used concurrently with approx/accu **/
void set_static_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch ((get_set_th(sp))->type) {
    case TYPE_I:   set_pr_threshold (sp, iter, attr);  break;
    case TYPE_II:  set_mg_threshold (sp, iter, attr); break;
    default:       error("set_static_threshold: no such type");  break;
  }
}


/** used concurrently with approx/accu **/
void set_pr_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ float this = (float) get_predicted_glub (sp, iter, attr);
  float prev = (float) sp->brother->glub;
  float r = get_r_factor (sp);
  float shed;
  double num, den;

  num = log10 ((double) (1.0 - cmd.cut_ratio + cmd.cut_ratio * r));
  den = log10 ((double) r);

  shed = (prev + (this - prev) * num / den) / (1.0 + sp->approx.approx);

  if (sp->constr.bound > shed) sp->constr.bound = shed;
  put_threshold(sp, sp->constr.bound);
}


/** used concurrently with approx/accu **/
void set_mg_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ search_ *brother = sp->brother;
  domain prev_th = get_threshold (sp->brother);
  domain prev_glub = sp->brother->glub;
  float g = get_g_factor (sp);
  float up, low, delta;

  low = tofloat (prev_glub) / (1.0 + brother->approx.approx);
  up = tofloat (prev_glub) / (1.0 + sp->approx.approx);
  delta = up - low;

  if (prev_th < low) sp->constr.bound = low + iter * g * delta;
  else sp->constr.bound = prev_th + g * delta;

  if (sp->constr.bound > up) sp->constr.bound = up;
  put_threshold (sp, sp->constr.bound);
}


void set_predictive_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{
  switch ((get_set_th(sp))->type) {
    case TYPE_I:   set_fr_threshold(sp, iter, attr);  break;
    case TYPE_II:  set_ld_threshold(sp, iter, attr); break;
    default:       error("set_predictive_threshold: no such type"); break;
  }
}


/** th vs time **/
void set_fr_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  float g = get_g_factor (sp);
  long this_time, time;
  double beta[2];

  /** if everything is too tough, simply use it **/
  sp->constr.bound = lb + iter * g * (ub - lb);

  /** if it is the last one, then use prediction **/
  if (attr == LAST_ITER) {
    if (get_th_vs_time (sp, beta, PARTIAL_SPAN) == YES) {
      time = sp->constr.time;
      sp->constr.bound = beta[0] + beta[1] * log10 ((double) time);
      if (sp->constr.bound <= sp->brother->constr.bound)
          sp->constr.bound = sp->brother->constr.bound + g * (ub - lb);
    }
  }

  else if (get_th_vs_time (sp, beta, FULL_SPAN) == YES) {
    this_time = get_r_factor (sp) * get_time (sp->brother);
    if (this_time > sp->constr.time) this_time = sp->constr.time;
    sp->constr.bound = beta[0] + beta[1] * log10 ((double) this_time);
      if (sp->constr.bound < sp->brother->constr.bound)
        sp->constr.bound = sp->brother->constr.bound + g * (ub - lb);
  }

  put_threshold (sp, sp->constr.bound);
}


/** lowb's bin vs time **/
void set_ld_threshold (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ search_ *brother = sp->brother;
  domain lb = sp->parent->root_lb;
  domain ub = sp->parent->root_ub;
  long this_time, time;
  double beta[2];
  float g = get_g_factor (sp);

  /** if everything is too tough, then simply use it **/
  sp->constr.bound = lb + iter * g * (ub - lb);

  /** if it is the last one, then use prediction **/
  if (attr == LAST_ITER) {
    if (get_bin_vs_time(sp, beta, PARTIAL_SPAN) == YES) {
      time = sp->constr.time;
      sp->constr.bound = beta[0] + beta[1] * log10 ((double) time);
      if (sp->constr.bound < sp->brother->constr.bound)
        sp->constr.bound = sp->brother->constr.bound + g * (ub - lb);
    }
  }

  else if (get_bin_vs_time (sp, beta, FULL_SPAN) == YES) {
      this_time = get_r_factor (sp) * get_time (brother);
      if (this_time > sp->constr.time) this_time = sp->constr.time;
      sp->constr.bound = beta[0] + beta[1] * log10 ((double) this_time);
      if (sp->constr.bound <= sp->brother->constr.bound)
        sp->constr.bound = sp->brother->constr.bound + g * (ub - lb);
  }

  put_threshold(sp, sp->constr.bound);
}



/*+ ------------------------------------------------- +
  +             RTS*    Library Routines              +
  + ------------------------------------------------- +*/


set_th_ *get_set_th (sp)
  search_ *sp;
{
  switch (sp->algorithm) {
    case uRTS:  return sp->alg.urts->set_th;    break;
    case bRTS:  return sp->alg.brts->set_th;    break;
    case hRTS:  return sp->alg.hrts->set_th;    break;
    case uIRD:          return sp->alg.uird->set_th;    break;
    case bIRD:          return sp->alg.bird->set_th;    break;
    default:            error("get_set_th: no such algorithm"); break;
  }
  return NULL;
}


domain get_predicted_glub (sp, iter, attr)
  search_ *sp;
  int iter;
  iter_attr_ attr;
{ span_ span = (attr == LAST_ITER) ? PARTIAL_SPAN : FULL_SPAN;
  float step, this_time;
  double beta[2];
  domain prev, prev_prev;

  if (get_glub_vs_time (sp, beta, span) == YES) {
    this_time = get_r_factor(sp) * get_time(sp->brother);
    return ((domain) (beta[0] + beta[1] * log10 ((double) this_time)));
  }

  /* if situation is too tough to handle, simply use naive glub */
  if (iter == 1) {
    step = get_g_factor(sp) * (sp->parent->root_ub - sp->parent->root_lb);
    this_time = sp->parent->root_ub - step;
  } else {
    prev = sp->brother->glub;
    if (sp->brother->brother) prev_prev = sp->brother->brother->glub;
    else prev_prev = sp->parent->root_ub;
    this_time = prev - (prev_prev - prev);
  }
  return ((domain) this_time);
}


yesno_ get_bin_vs_time (sp, beta, span)
  search_ *sp;
  double beta[];
  span_ span;
{ search_ *brother = sp->brother;
  domain th, exp_opt, *calibrate;
  set_th_ *set_th;
  long *bin, num;
  double x_sum = 0.0, y_sum = 0.0, x2_sum = 0.0, xy_sum = 0.0, x, y, det;
  int nbin, n, i;

  if (brother == NULL) return NO;

  if (is_feasible_time (get_time (brother))) {
    th = get_threshold (brother);
    exp_opt = (domain) (brother->glub / (1.0 + get_approx(brother)));
    th = min (th, exp_opt);

    set_th = get_set_th (brother);
    nbin = set_th->nbin;
    bin = set_th->bin;
    calibrate = set_th->calibrate;

    /* first-order regression over bins, x is #nodes, y is lowb val */
    for (n = i = 0; i < nbin && *(calibrate+i) <= th; i++) {
      num = *(bin+i);
      if (num < MinValidBin) continue;
      n++;
      x = log10 ((double) num);
      y = (double) *(calibrate+i);
      x_sum += x;
      y_sum += y;
      x2_sum += (x * x);
      xy_sum += (x * y);
    }

    if (n <= 1) return NO;
    det = todouble (n) * x2_sum - x_sum * x_sum;
    if (tofloat (det) == 0.0) return NO;
    beta[0] = (x2_sum * y_sum - x_sum * xy_sum) / det;
    beta[1] = (- x_sum * y_sum + todouble (n) * xy_sum) / det;

    if (tofloat (beta[1]) == 0.0) return NO;
    return YES;
  }

  return NO;
}


yesno_ get_th_vs_time (sp, beta, span)
  search_ *sp;
  double beta[];
  span_ span;
{ search_ *brother;
  double x_sum = 0.0, y_sum = 0.0, x2_sum = 0.0, xy_sum = 0.0, x, y, det;
  long time;
  int n = 0;

  /* skip infeasible regression */
  if (check_span (sp, span) == NO) return NO;

  /* first-order regression over bins, x is #nodes, y is lowb val */
  for (brother = sp->brother; brother; brother = brother->brother) {
    time = get_time (brother);
    if (is_feasible_time (time)) {
      n++;
      x = log10 ((double) time);
      y = (double) get_threshold (brother);
      x_sum += x;
      y_sum += y;
      x2_sum += (x * x);
      xy_sum += (x * y);
    } else break;
  }

  det = todouble (n) * x2_sum - x_sum * x_sum;
  if (tofloat (det) == 0.0) return NO;
  beta[0] = (x2_sum * y_sum - x_sum * xy_sum) / det;
  beta[1] = (- x_sum * y_sum + todouble (n) * xy_sum) / det;

  if (tofloat (beta[1]) == 0.0) return NO;
  return YES;
}


yesno_ get_glub_vs_time (sp, beta, span)
  search_ *sp;
  double beta[];
  span_ span;
{ search_ *brother;
  double x_sum = 0.0, y_sum = 0.0, x2_sum = 0.0, xy_sum = 0.0, x, y, det;
  long time;
  int n = 0;

  /* skip infeasible regression */
  if (check_span (sp, span) == NO) return NO;

  /* first-order regression over bins, x is #nodes, y is lowb val */
  for (brother = sp->brother; brother; brother = brother->brother) {
    time = get_time (brother);
    if (is_feasible_time (time)) {
      n++;
      x = log10 ((double) time);
      y = (double) brother->glub;
      x_sum += x;
      y_sum += y;
      x2_sum += (x * x);
      xy_sum += (x * y);
    } else break;
  }

  det = todouble (n) * x2_sum - x_sum * x_sum;
  if (tofloat (det) == 0.0) return NO;
  beta[0] = (x2_sum * y_sum - x_sum * xy_sum) / det;
  beta[1] = (- x_sum * y_sum + todouble (n) * xy_sum) / det;

  if (tofloat (beta[1]) == 0.0) return NO;
  return YES;
}


yesno_ get_approx_vs_time (sp, beta, span)
  search_ *sp;
  double beta[];
  span_ span;
{ search_ *brother;
  double x_sum = 0.0, y_sum = 0.0, x2_sum = 0.0, xy_sum = 0.0;
  double x, y, det;
  long time;
  int n = 0;

  /* skip infeasible regression */
  if (check_span (sp, span) == NO) return NO;

  /* first-order regression over bins, x is #nodes, y is lowb val */
  for (brother = sp->brother; brother; brother = brother->brother) {
    time = get_time (brother);
    if (is_feasible_time (time)) {
      n++;
      x = log10 ((double) time);
      y = (double) get_approx (brother);
      x_sum += x;
      y_sum += y;
      x2_sum += (x * x);
      xy_sum += (x * y);
    } else break;
  }

  det = todouble (n) * x2_sum - x_sum * x_sum;
  if (tofloat (det) == 0.0) return NO;
  beta[0] = (x2_sum * y_sum - x_sum * xy_sum) / det;
  beta[1] = (- x_sum * y_sum + todouble (n) * xy_sum) / det;

  if (tofloat (beta[1]) == 0.0) return NO;
  return YES;
}


yesno_ get_accu_vs_time (sp, beta, span)
  search_ *sp;
  double beta[];
  span_ span;
{ search_ *brother;
  double x_sum = 0.0, y_sum = 0.0, x2_sum = 0.0, xy_sum = 0.0;
  double x, y, det;
  long time;
  int n = 0;
  float a;

  /* skip infeasible regression */
  if (check_span (sp, span) == NO) return NO;

  /* first-order regression over bins, x is #nodes, y is lowb val */
  for (brother = sp->brother; brother; brother = brother->brother) {
    time = get_time (brother);
    if (is_feasible_time (time)) {
      n++;
      a = get_approx (brother);
      a = approx2accu (a);
      x = log10 ((double) time);
      y = todouble (a);
      x_sum += x;
      y_sum += y;
      x2_sum += (x * x);
      xy_sum += (x * y);
    } else break;
  }
```

```
  det = todouble (n) * x2_sum - x_sum * x_sum;
  if (tofloat (det) == 0.0) return NO;
  beta[0] = (x2_sum * y_sum - x_sum * xy_sum) / det;
  beta[1] = (- x_sum * y_sum + todouble (n) * xy_sum) / det;

  if (tofloat (beta[1]) == 0.0) return NO;
  return YES;
}


yesno_ get_glub_vs_approx (sp, beta, span)
  search_ *sp;
  double beta[];
  span_ span;
{ search_ *brother;
  double x_sum = 0.0, y_sum = 0.0, x2_sum = 0.0, xy_sum = 0.0, x, y, det;
  int n = 0;

  /* skip infeasible regression */
  if (check_span (sp, span) == NO) return NO;

  /* first-order regression over bins, x is #nodes, y is lowb val */
  for (brother = sp->brother; brother; brother = brother->brother)
    if (is_feasible_time (get_time (brother))) {
      n++;
      x = (double) brother->approx.approx;
      y = (double) brother->glub;
      x_sum += x;
      y_sum += y;
      x2_sum += (x * x);
      xy_sum += (x * y);
    } else break;

  det = todouble (n) * x2_sum - x_sum * x_sum;
  if (tofloat (det) == 0.0) return NO;
  beta[0] = (x2_sum * y_sum - x_sum * xy_sum) / det;
  beta[1] = (- x_sum * y_sum + todouble (n) * xy_sum) / det;

  if (tofloat (beta[1]) == 0.0) return NO;
  return YES;
}


yesno_ check_span (sp, span)
  search_ *sp;
  span_ span;
{ search_ *brother;
  int count = 0;

  for (brother = sp->brother; brother; brother = brother->brother)
    if (is_feasible_time (get_time (brother))) count++;
    else break;

  if (span == FULL_SPAN) return (count >= 2 ? YES : NO);
  else return (count >= 3 ? YES : NO);
}


void reset_th_environ (sp)
  search_ *sp;
{ set_th_ *set_th = get_set_th(sp);

  /* reset environ for predicting threshold */
  switch (set_th->alg) {
    case NAIVE_TH:                             break;
    case STATIC_TH:                            break;
    case PREDICTIVE_TH: reset_th_bin(sp);      break;
    default: error("set_rts_threshold: no such threshold setting"); break;
  }
}


void reset_th_bin (sp)
  search_ *sp;
{ set_th_ *set_th = get_set_th(sp);
  int nbin = set_th->nbin, i;
  long *bin = set_th->bin;
  domain *calibrate = set_th->calibrate;
  domain abs_lb = sp->parent->root_lb;
  domain abs_ub = sp->parent->root_ub;
  float step;

  /* reset bin and set the calibration of bins */
  step = (float) (abs_ub - abs_lb);
  step /= tofloat (nbin);
  for (i = 0; i < nbin; i++) {
    *(bin+i) = 0;
    *(calibrate+i) = abs_lb + (domain) (step * i);
  }
}


void enter_th_bin (sp, lowb)
  search_ *sp;
  domain lowb;
{ set_th_ *set_th = get_set_th(sp);
  int nbin = set_th->nbin, i;
  domain *calibrate = set_th->calibrate;

  for (i = 0; i < nbin && *(calibrate+i) < lowb; i++) ;
  if (i == nbin) i = nbin - 1;
  ++(*(set_th->bin+i));
}


void set_th_entry (sp, node)
  search_ *sp;
  node_ *node;
{
  switch ((get_set_th(sp))->alg) {
    case NAIVE_TH:                                     break;
    case STATIC_TH:                                    break;
    case PREDICTIVE_TH: enter_th_bin(sp, node->lowb);  break;
    default: error("set_th_entry: no such threshold algorithm"); break;
  }
}


float get_g_factor (sp)
  search_ *sp;
{ float g;
  switch (sp->algorithm) {
    case uRTS:  g = sp->alg.urts->g_factor;     break;
    case bRTS:  g = sp->alg.brts->g_factor;     break;
    case hRTS:  g = sp->alg.hrts->g_factor;     break;
    case uIRD:  g = sp->alg.uird->g_factor;     break;
    case bIRD:  g = sp->alg.bird->g_factor;     break;
    default:  error("get_g_factor: no such algorithm"); break;
  }

#ifdef GRADIENT_STEP
  if (g == 0.0 || g == huge_float) g = 0.1;
  g = 1.0 / g;
#else
  if (g == huge_float) g = 0.1;
#endif
  return g;
}


float get_r_factor (sp)
  search_ *sp;
{ search_ *p;
  float r_sum = 0.0, r;
  int count;

  switch (sp->algorithm) {
    case uRTS:  r =  sp->alg.urts->r_factor;    break;
    case bRTS:  r =  sp->alg.brts->r_factor;    break;
    case hRTS:  r =  sp->alg.hrts->r_factor;    break;
    case uIRD:  r =  sp->alg.uird->r_factor;    break;
    case bIRD:  r =  sp->alg.bird->r_factor;    break;
    default:  error("get_r_factor: no such algorithm"); break;
  }

  if (r == huge_float) { /* growth ratio is not specified */
    if ((p = sp->brother))
      for (count = 0; p->brother; p = p->brother, count++)
        r_sum += (((float) get_time(p)) / ((float) get_time(p->brother)));
    r = count ? r_sum / ((float) count) : 2;
  }

  return r;
}


iter_attr_ get_iteration_attribute (sp, iter)
  search_ *sp;
  int iter;
{ long prev_time, this_time, curr_time, Tconstr;
  float r = get_r_factor (sp);

  if (iter > 1) {
    prev_time = get_time(sp->brother);
    this_time = r * prev_time;
    curr_time = get_time(sp->parent);
    Tconstr = sp->parent->constr.time;
    if (curr_time + 2 * this_time >= Tconstr) return LAST_ITER;
    if (curr_time + (1+r) * this_time >= Tconstr) return LAST_2ND_ITER;
  }

  return OTHER_ITER;
}

::::::::::::::
algorithm/tca.c
::::::::::::::

/* Time-Constrained A* search algorithms.
 * TCA=<sTCA,pTCA,dTCA>
 */
```

```
/* sTCA: stca_algorithm(search_ *sp) */
void stca_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  int n = 0;

  sp->approx.achieved = sp->approx.root_approx;
  child_sp = create_child_search(sp);
  put_child_constr(child_sp, sp, 1.0, 1.0, 1.0);
  set_search_message(&msg, FRESH_START, NULL, NO, cmd.pf.rt, cmd.pf.st);
  while (1) {
    child_sp->approx.approx = get_stca_next_approx(sp, ++n);
    if (cmd.xon == YES)
      child_sp->approx.accu = approx2accu (child_sp->approx.approx);
    if (bfs_primitive(child_sp, sp, &msg) == SEARCH_IS_ABORTED) break;
    eval_final_approx(child_sp);
    merge_stat_to_parent(child_sp, sp);
    merge_approx_to_parent(child_sp, sp);
    merge_sol_to_parent(child_sp, sp);
    if (child_sp->approx.approx <= 0.0) break;
    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
    put_child_constr(child_sp, sp, 1.0, 1.0, 1.0);
    inherit_sol_from_parent(child_sp, sp);
  }
  merge_stat_to_parent(child_sp, sp);
  merge_sol_to_parent(child_sp, sp);
  if (get_approx(child_sp) < sp->approx.achieved) {
    merge_approx_to_parent(child_sp, sp);
  }
}


float get_stca_next_approx (sp, n)
/* a = a0 * (1 - n * g); a = (a < 0) ? 0 : a; */
  search_ *sp;
  int n;
{ float new_a;

  if (cmd.xon == YES) {
    /* accuracy-driven */
    new_a = sp->approx.root_accu *
                                  (1.0 - tofloat (n) * sp->alg.stca->g_factor);
    new_a = accu2approx (new_a);
  } else {
    /* approx-driven */
    new_a = sp->approx.root_approx *
                                  (1.0 - tofloat (n) * sp->alg.stca->g_factor);
  }
  if (new_a < 0.0) new_a = 0.0;
  return new_a;
}


/* pTCA: ptca_algorithm(search_ *sp) */
void ptca_algorithm (sp)
  search_ *sp;
{ constr_ save_constr;
  search_message_ msg;
  ptca_ *ptcap = sp->alg.ptca;
  float s;

  sp->approx.achieved = sp->approx.root_approx;
  s = ptcap->s_factor;
  init_regress_struct(&(ptcap->regress));
  set_search_message(&msg, FRESH_START, &(ptcap->regress), NO,
                                    cmd.pf.rt, cmd.pf.st);
  save_constr = sp->constr;
  /* nTCA & profiling parts */
  set_stop_constr(sp, s, s, s);
  bfs_primitive(sp, NULL, &msg);
  /* prediction part */
  alg_pf_predict(sp, &(ptcap->regress));
  sp->approx.approx = sp->approx.predicted * ptcap->c_factor;
  /* solution part */
  sp->constr = save_constr;
  msg.style = RESUME;
  msg.rp = NULL;
  bfs_primitive(sp, NULL, &msg);
  eval_final_approx(sp);
}


/* dTCA: dtca_algorithm(search_ *sp) & dtca_entry(search_ *sp) */
void dtca_algorithm (sp)
  search_ *sp;
{ constr_ save_constr;
  search_message_ msg;
  dtca_ *dtcap = sp->alg.dtca;
  float s;

  sp->approx.achieved = sp->approx.root_approx;
  s = dtcap->s_factor;
  init_regress_struct(&(dtcap->regress));
  save_constr = sp->constr;
  set_stop_constr(sp, s, s, s);
  set_search_message(&msg, FRESH_START, &(dtcap->regress), NO,
                                cmd.pf.rt, cmd.pf.st);
  bfs_primitive(sp, NULL, &msg);
  alg_pf_predict(sp, &(dtcap->regress));
  sp->approx.approx = sp->approx.predicted * dtcap->c_factor;
  sp->constr = save_constr;
  msg.style = RESUME;
  msg.alg_entry = YES;
  bfs_primitive(sp, NULL, &msg);
  eval_final_approx(sp);
}


void dtca_entry(sp)
  search_ *sp;
{ float temp;

  alg_pf_predict(sp, &(sp->alg.dtca->regress));
  temp = sp->approx.predicted * sp->alg.dtca->c_factor;
  /* approx can only go up and can not go down */
  if (sp->approx.approx < temp) sp->approx.approx = temp;
}

::::::::::::::
algorithm/tcgd.c
::::::::::::::

/* Time-Constrained GDFS algorithms.
 * TCGD=<sTCGD,pTCGD>
 */
```

```
/* sTCGD: stcgd_algorithm(search_ *sp) */
void stcgd_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  search_message_ msg;
  int n = 0;

  sp->approx.achieved = sp->approx.root_approx;
  child_sp = create_child_search(sp);
  put_child_constr(child_sp, sp, 1.0, 1.0, 1.0);
  set_search_message(&msg, FRESH_START, NULL, NO, cmd.pf.rt, cmd.pf.st);
  while (1) {
    child_sp->approx.approx = get_stcgd_next_approx(sp, ++n);
    if (cmd.xon == YES)
      child_sp->approx.accu = approx2accu (child_sp->approx.approx);
    if (gdfs_primitive(child_sp, sp, &msg) == SEARCH_IS_ABORTED) break;
    eval_final_approx(child_sp);
    merge_stat_to_parent(child_sp, sp);
    merge_approx_to_parent(child_sp, sp);
    merge_sol_to_parent(child_sp, sp);
    if (child_sp->approx.approx <= 0.0) break;
    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
    put_child_constr(child_sp, sp, 1.0, 1.0, 1.0);
    inherit_sol_from_parent(child_sp, sp);
  }
  merge_stat_to_parent(child_sp, sp);
  merge_sol_to_parent(child_sp, sp);
  if (get_approx(child_sp) < sp->approx.achieved) {
    merge_approx_to_parent(child_sp, sp);
  }
}


float get_stcgd_next_approx (sp, n)
/* a = a0 * (1 - n * g); a = (a < 0) ? 0 : a; */
  search_ *sp;
  int n;
{ float new_a;

  if (cmd.xon == YES) {
    /* accuracy-driven */
    new_a = sp->approx.root_accu *
                                  (1.0 - tofloat (n) * sp->alg.stcgd->g_factor);
    new_a = accu2approx (new_a);
  } else {
    /* approx-driven */
    new_a = sp->approx.root_approx *
                                  (1.0 - tofloat (n) * sp->alg.stcgd->g_factor);
  }
  if (new_a < 0.0) new_a = 0.0;
  return new_a;
}


/* pTCGD: ptcgd_algorithm(search_ *sp) */
void ptcgd_algorithm (sp)
  search_ *sp;
{ search_ *child_sp;
  constr_ save_constr;
  search_message_ msg;
  ptcgd_ *ptcgdp = sp->alg.ptcgd;
  float s;
  int n = 0;

  sp->approx.achieved = sp->approx.root_approx;
  s = ptcgdp->s_factor;
  init_regress_struct(&(ptcgdp->regress));
  set_search_message(&msg, FRESH_START, NULL, NO, cmd.pf.rt, cmd.pf.st);
  /* create search struct for sTCGD */
  child_sp = create_child_search(sp);
  put_child_constr(child_sp, sp, s);
  /* sTCGD & profiling part */
  while (1) {
    child_sp->approx.approx = get_stcgd_next_approx(sp, ++n);
    if (cmd.xon == YES)
      child_sp->approx.accu = approx2accu (child_sp->approx.approx);
    if (gdfs_primitive(child_sp, sp, &msg) == SEARCH_IS_ABORTED) break;
    eval_final_approx(child_sp);
    merge_stat_to_parent(child_sp, sp);
    merge_approx_to_parent(child_sp, sp);
    merge_sol_to_parent(child_sp, sp);
    if (child_sp->approx.approx <= 0.0) break;
    alg_pf(child_sp, &(sp->alg.ptcgd->regress));
    save_constr = child_sp->constr;
    (sp->child = child_sp = create_brother_search(child_sp))->parent = sp;
    child_sp->constr = save_constr;
    inherit_sol_from_parent(child_sp, sp);
  }
  merge_stat_to_parent(child_sp, sp);
  if (get_approx(child_sp) < sp->approx.achieved) {
    merge_approx_to_parent(child_sp, sp);
    merge_sol_to_parent(child_sp, sp);
  }
  /* prediction part */
  alg_pf_predict(sp, &(ptcgdp->regress));
  sp->approx.approx = sp->approx.predicted * ptcgdp->c_factor;
  /* solution part */
  gdfs_primitive(sp, NULL, &msg);
  eval_final_approx(sp);
}
```

```
Thu Jan 30 15:32:27 CST 1992
::::::::::::::
primitive/band.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ band_primitive (sp, offset_sp, msgp)
    search_ *sp, *offset_sp;
    search_message_ *msgp;
{   node_ *node, *children, *child, *p, *temp, *temp0, *next_child, *new_list;
    int i;
    solution_ *sol;
    yesno_ bound_dom, to_update_gllb;
    float GUIDE_H ();
#ifdef HARE_GUIDANCE
    domain g_cost;
#endif
#ifdef DEBUG
if (cmd.debug >= 2) printf ("\nenter band_primitive\n");
if (cmd.debug >= 1) printf ("bw_fx=%s\n", dbg_bw_fx[cmd.bw_fx]);
#endif

    times (&start_time);
    last_time = start_time;
    if (msgp->style == FRESH_START) {
        idpd_init ();
        put_band_node (create_root (sp), sp);
    }

    while ((node = get_band_node (sp))) {

#ifdef HARE_GUIDANCE
        g_cost = node->g_cost;
#endif

        if (is_bounded (node, sp)) {
            sp->stat.bounded++;
            free_node (node);
            continue;
        }

        to_update_gllb = (sp->gllb == node->lowb) ? YES : NO;
        eval_rt_approx (sp);
        update_stat (sp);
        if (msgp->rt_pf == YES) pf_run_time (sp, offset_sp);
        if (msgp->st_pf == YES) pf_space_vs_time (sp, offset_sp);
        if (is_constr_violated (sp)) return SEARCH_IS_ABORTED;
        if (msgp->rp) alg_rt_pf (sp, msgp->rp);
        if (msgp->alg_entry == YES) alg_entry (sp, node);

#ifdef DEBUG
if (cmd.debug >= 2) debug_node ("\nexpand", node, sp);
#endif

        /** expand will return a list of children **/
        if ((children = expand (node, ALL_CHILDREN, DONT_CARE)))
            sp->stat.expanded++;
        free_node (node);

#if defined (LOWB_GUIDANCE) || defined (UPB_GUIDANCE) || defined (UGDFS) || defined (HAR
E_GUIDANCE)
#define BOUND_GUIDANCE
#endif

        new_list = NULL;
        for (child = children; child; child = next_child) {
            sp->stat.generated++;
            next_child = child->brother;

            bound_dom = YES;    /** init skip **/

            if (is_infeasible (child))  {
#ifdef DEBUG
if (cmd.debug >= 3) printf ("infeasible\n");
#endif
              sp->stat.infeasible++;
              free_node (child);
              continue;
            }

#ifdef DEBUG
if (cmd.debug >= 3) printf ("eval bounds\n");
#endif

            evaluate_lower_bound (child);
            sol = evaluate_upper_bound (child, get_sol_buf ());

#ifdef DEBUG
            if (cmd.debug >= 3) debug_node (NULL, child, sp);
#endif

            if (child->upb < sp->glub) {
#ifdef DEBUG
if (cmd.debug >= 3) printf ("update incumbent\n");
#endif
                sp->glub = child->upb;
                eval_rt_approx (sp);
                free_sol_buf (sp->incumbent);
                sp->incumbent = sol;
                bounding (sp);
                dominating (child, sp);
                bound_dom = NO; /** skip over  bounded & dominated  tests **/
            }
            else free_sol_buf (sol);

            if (is_feasible_or_equiv (child)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf ("feasible\n");
#endif
                sp->stat.feasible++;
                free_node (child);
                continue;
            }

            if (is_bounded (child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf ("bounded\n");
#endif
                sp->stat.bounded++;
                free_node (child);
                continue;
            }

            if (bound_dom == YES) {
                if (is_dominated (child, sp)) {
#ifdef DEBUG
```

```
if (cmd.debug >= 3) printf ("dominated\n");
#endif
                    sp->stat.dominated++;
                    free_node (child);
                    continue;
                }
            }

            if (is_hard_bounded (child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf ("bounded\n");
#endif
                if (Next_IDA_Threshold > child->lowb)
                    Next_IDA_Threshold = child->lowb;
                sp->stat.hard_bounded++;
                free_node(child);
                continue;
            }

            /** by non-leaf, lousy-upper-bound node **/
            if (bound_dom == YES) dominating (child, sp);

            child->next = new_list;
            new_list = child;

        } /** children loop **/

        children = new_list;
        new_list = NULL;

#ifdef BOUND_GUIDANCE
        /** sort these live children by fancy guidance **/
        for (child = children; child; child = child->next) {
            if (new_list) {
                for (temp0 = temp = new_list;
                     temp;
                     temp = (temp0 = temp)->brother) {

#if defined (HARE_GUIDANCE)
                    if (GUIDE_H (child->upb, child->lowb, child->depth,
                                 child->g_cost - g_cost) <
                        GUIDE_H (temp->upb, temp->lowb, temp->depth,
                                 temp->g_cost - g_cost)) {
#elif defined (UPB_GUIDANCE) || defined (UGDFS)
                    if (child->upb < temp->upb) {
#elif defined (LOWB_GUIDANCE)
                    if (child->lowb < temp->lowb) {
#else /** bad guidance **/
                    if (1) {
#endif

                        if (temp == new_list) (new_list = child)->brother = temp;
                        else (temp0->brother = child)->brother = temp;
                        break;
                    }
                }
                if (! temp) (temp0->brother = child)->brother = NULL;
            }
            else (new_list = child)->brother = NULL;
        }
        children = new_list;

#else DEFAULT_GUIDANCE
        for (child = children; child; child = child->next) {
            child->brother = new_list;
```

```
            new_list = child;
        }
        children = new_list;

#endif BOUND_GUIDANCE


        for (child = children; child; child = child->brother)
            child->next = NULL;

        put_band_node (children, sp);

        if (to_update_gllb == YES) {
            /** set init value for global_lowb **/
            sp->gllb = sp->glub;
            /** check band priority list **/
            if (sp->open.btree)
                /** global lowb == min { lowb of all active nodes } **/
                for (p = sp->open.btree; p; p = p->next)
                    if (p->lowb < sp->gllb) sp->gllb = p->lowb;
            /** check young priority lists **/
            for (i = sp->band.ceiling_depth; i <= sp->band.floor_depth; i++)
                if ((sp->band.tree+i)->young)
                    /** global lowb == min { lowb of all active nodes } **/
                    for (p = (sp->band.tree+i)->young; p; p = p->next)
                        if (p->lowb < sp->gllb) sp->gllb = p->lowb;
        }

    } /** selection loop **/

    /** When the search is completed, the global lowb and the global upb
     ** should be the same, regardless of approximation is used or not.
     ** However, it is not the case, sometimes.
     ** One possible reason is that
     ** global lowb is supposed to be updated by a feasible sol; however,
     ** the feasible sol is freed during the children loop.
     ** Therefore, both are coersed to be equal.
     **/
    sp->gllb = sp->glub;
    /** if (sp->approx.approx == 0.0) sp->gllb = sp->glub; **/

    times (&break_time);
    sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
    sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

    if (sp->constr.bound != problem.huge)
        if (sp->glub <= sp->constr.bound) msgp->feasible = YES;

    return SEARCH_IS_COMPLETED;
} /** band primitive **/


int bandwidth_function (depth, init_width)
    int depth, init_width;
{   int width = init_width;
    static float a, b;
    static int first_entry = 1;

    if (first_entry) {
        first_entry = 0;
        switch (cmd.bw_fx) {
        case LINEAR_BW:
                a = ((float) (1 - init_width)) / (problem.size - 1);
                b = init_width - a;
                break;

        case EXP_BW:
                a = ((float) (log ((double) init_width))) / (problem.size - 1);
                break;
        default: break;
        }
    }

    switch (cmd.bw_fx) {
    case LINEAR_BW:
                width = a * depth + b;
                break;

    case EXP_BW:
                width = init_width * exp ((double) (a * (1 - depth)));
                break;

    case FIXED_BW:
    default:    break;
    }

    if (width < 1) width = 1;
    return width;
}

::::::::::::::
primitive/bfs.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ bfs_primitive (sp, offset_sp, msgp)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
{ node_ *node, *children, *child, *new_list;
  solution_ *sol;
  yesno_ bound_dom;
#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter bfs_primitive\n");
#endif

  times(&start_time);
  last_time = start_time;
  if (msgp->style == FRESH_START) {
    idpd_init();
    bptree_insert(create_root(sp), sp);
  }

  while ((node = bptree_delete(sp))) {

    if (is_bounded(node, sp)) {
      sp->stat.bounded++;
      free_node(node);
      continue;
    }

    sp->gllb = node->lowb;
    eval_rt_approx(sp);
    update_stat(sp);

    if (msgp->rt_pf == YES) pf_run_time(sp, offset_sp);
    if (msgp->st_pf == YES) pf_space_vs_time(sp, offset_sp);
```

```
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    if (msgp->rp) alg_rt_pf(sp, msgp->rp);
    if (msgp->alg_entry == YES) alg_entry(sp, node);
#ifdef DEBUG
if (cmd.debug >= 2) debug_node("\nexpand", node, sp);
#endif
    /* expand will return a list of children */
    if ((children = expand(node, ALL_CHILDREN, DONT_CARE)))
      sp->stat.expanded++;
    free_node(node);
    new_list = NULL;

    for (child = children; child; child = child->brother) {
      sp->stat.generated++;

      if (is_infeasible(child))  {
#ifdef DEBUG
if (cmd.debug >= 3) printf("infeasible\n");
#endif
        sp->stat.infeasible++;
        free_node(child);
        continue;
      }

#ifdef DEBUG
if (cmd.debug >= 3) printf("eval bounds\n");
#endif

      evaluate_lower_bound(child);
      sol = evaluate_upper_bound(child, get_sol_buf());

#ifdef DEBUG
if (cmd.debug >= 3) debug_node(NIL, child, sp);
#endif
      bound_dom = YES;  /* init skip */

      if (child->upb < sp->glub) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("update incumbent\n");
#endif
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bounding(sp);
        dominating(child, sp);
        bound_dom = NO; /* skip over  bounded & dominated  tests */
      }
      else free_sol_buf(sol);

      if (is_feasible_or_equiv(child)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("feasible\n");
#endif
        sp->stat.feasible++;
        free_node(child);
        continue;
      }

      if (is_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
          sp->stat.bounded++;
          free_node(child);
          continue;
      }

      if (bound_dom == YES) {
        if (is_dominated(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("dominated\n");
#endif
          sp->stat.dominated++;
          free_node(child);
          continue;
        }
      }

      if (is_hard_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        sp->stat.hard_bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES)
        dominating(child, sp);  /* by non-leaf, lousy-upper-bound node */

      if (new_list) {
        child->next = new_list;
        new_list = child;
      }
      else (new_list = child)->next = NULL;
    } /* children loop */

    for (child = new_list; child; child = child->next)
      child->brother = child->next;

    for (child = new_list; child; child = child->brother) {
      child->next = NULL;
      bptree_insert(child, sp);
    }
  } /* selection loop */

  /* When the search is completed, the global lowb and the global upb
   * should be the same, regardless of approximation is used or not.
   * However, it is not the case, sometimes.
   * One possible reason is that
   * global lowb is supposed to be updated by a feasible sol; however,
   * the feasible sol is freed during the children loop.
   * Therefore, both are coersed to be equal.
   */
  /* not true for BFS, sp->gllb = sp->glub; */
  /** true for BFS, too **/
  sp->gllb = sp->glub;
  /** if (sp->approx.approx == 0.0) sp->gllb = sp->glub; **/

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

  if (sp->constr.bound != problem.huge)
    if (sp->glub <= sp->constr.bound) msgp->feasible = YES;

  return SEARCH_IS_COMPLETED;
```

```
} /* bfs primitive */

::::::::::::::
primitive/dfs.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ dfs_primitive (sp, offset_sp, msgp)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
{ node_ *node, *child;
  solution_ *sol;
  yesno_ to_update_gllb, first_expanded, bound_dom;

#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter dfs_primitive\n");
#endif

  times(&start_time);
  last_time = start_time;
  if (msgp->style == FRESH_START) {
    idpd_init();
    stack_push(create_root(sp), sp);
  }

  first_expanded = YES;
  /* explore problem */
  while ((node = stack_top(sp))) {

    if (is_bounded(node, sp)) { sp->stat.bounded++; free_node(node); continue; }

    to_update_gllb = (node->lowb == sp->gllb) ? YES : NO;
    update_stat(sp);
    if (msgp->rt_pf == YES) pf_run_time(sp, offset_sp);
    if (msgp->st_pf == YES) pf_space_vs_time(sp, offset_sp);
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    if (msgp->rp) alg_rt_pf(sp, msgp->rp);
    if (msgp->alg_entry == YES) alg_entry(sp, node);
#ifdef DEBUG
if (cmd.debug >= 2) debug_node("\nexpand", node, sp);
#endif
    /* sprout only one child at a time */
    child = expand(node, NEXT_CHILD, DONT_CARE);
    if (child == NIL) { stack_pop(sp); continue; }
    else sp->stat.generated++;
    if (first_expanded == YES) { sp->stat.expanded++; first_expanded = NO; }

    if (is_infeasible(child))  {
#ifdef DEBUG
if (cmd.debug >= 3) printf("infeasible\n");
#endif
      sp->stat.infeasible++;
      free_node(child);
      continue;
    }

#ifdef DEBUG
if (cmd.debug >= 3) printf("eval bounds\n");
#endif

    evaluate_lower_bound(child);
    sol = evaluate_upper_bound(child, get_sol_buf());
```

```
#ifdef DEBUG
if (cmd.debug >= 3) debug_node(NIL, child, sp);
#endif
      bound_dom = YES; /* init skip */

      if (child->upb < sp->glub) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("update incumbent\n");
#endif
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bound_dom = NO;       /* skip over  bounded & dominated  tests */
      }
      else free_sol_buf(sol);

    if (is_feasible_or_equiv(child)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("feasible\n");
#endif
      sp->stat.feasible++;
      free_node(child);
      continue;
    }

    if (bound_dom == YES) {
      if (is_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        sp->stat.bounded++;
        free_node(child);
        continue;
      }
    }

    if (is_hard_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
      if (Next_IDA_Threshold > child->lowb) Next_IDA_Threshold = child->lowb;
      sp->stat.hard_bounded++;
      free_node(child);
      continue;
    }

    stack_push(child, sp);
    first_expanded = YES;

    /* update gllb */
    if (to_update_gllb == YES) {
      sp->gllb = (is_stack_empty(sp)) ? sp->glub : (stack_bottom(sp))->lowb;
      eval_rt_approx(sp);
    }
  } /* selection loop */

  /* When the search is completed, the global lowb and the global upb
   * should be the same, regardless of approximation is used or not.
   * However, it is not the case, sometimes.
   * One possible reason is that
   * global lowb is supposed to be updated by a feasible sol; however,
   * the feasible sol is freed during the children loop.
   * Therefore, both are coersed to be equal.
   */
  sp->gllb = sp->glub;
  /** if (sp->approx.approx == 0.0) sp->gllb = sp->glub; **/

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

  if (sp->constr.bound != problem.huge)
    if (sp->glub <= sp->constr.bound) msgp->feasible = YES;

  return SEARCH_IS_COMPLETED;
} /* dfs primitive */

::::::::::::::
primitive/first.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ first_primitive (sp, offset_sp, msgp, num_active)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
  int num_active;
{ node_ *node, *children, *child, *temp;
  solution_ *sol;
  yesno_ bound_dom;
  long count;
#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter first_primitive\n");
#endif

  times(&start_time);
  last_time = start_time;
  if (msgp->style == FRESH_START) {
    idpd_init();
    btree_insert(create_root(sp), sp);
  }

  while (1) {
    for (count = 0, temp = sp->open.btree; temp; temp = temp->next)
      if (temp->type == ACTIVE_SINGLE) count++;
    if (count >= num_active) return SEARCH_IS_COMPLETED;
    if (! (node = btree_delete(sp))) break;
    sp->gllb = node->lowb;
    update_stat(sp);
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    /* expand will return a list of children */
    if ((children = expand(node, ALL_CHILDREN, DONT_CARE))) sp->stat.expanded++;
    free_node(node);

    for (child = children; child; child = child->brother) {
      sp->stat.generated++;

      if (is_infeasible(child))  {
        sp->stat.infeasible++;
        free_node(child);
        continue;
      }

      evaluate_lower_bound(child);
      sol = evaluate_upper_bound(child, get_sol_buf());
      bound_dom = YES;  /* init skip */

      if (child->upb < sp->glub) {
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bounding(sp);
        dominating(child, sp);
        bound_dom = NO; /* skip over  bounded & dominated  tests */
      }
      else free_sol_buf(sol);

      if (is_feasible_or_equiv(child)) {
        sp->stat.feasible++;
        free_node(child);
        continue;
      }

      if (is_bounded(child, sp)) {
        sp->stat.bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES) {
        if (is_dominated(child, sp)) {
          sp->stat.dominated++;
          free_node(child);
          continue;
        }
      }

      if (is_hard_bounded(child, sp)) {
        sp->stat.hard_bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES)
        dominating(child, sp);  /* by non-leaf, lousy-upper-bound node */

      btree_insert(child, sp);
    } /* children loop */
  } /* selection loop */

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

  return SEARCH_IS_COMPLETED;
} /* first primitive */
::::::::::::::
primitive/gbb.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ gbb_primitive (sp, offset_sp, msgp)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
{ node_ *node, *children, *child;
  solution_ *sol;
  yesno_ bound_dom;
#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter gbb_primitive\n");
#endif

  times(&start_time);  last_time = start_time;
  if (msgp->style == FRESH_START) {
    idpd_init();
    list_insert(create_root(sp), sp);
  }

  while ((node = list_delete(sp))) {

    if (is_bounded(node, sp)) { sp->stat.bounded++; free_node(node); continue; }

    sp->gllb = node->lowb;
    eval_rt_approx(sp);
    update_stat(sp);
    if (msgp->rt_pf == YES) pf_run_time(sp, offset_sp);
    if (msgp->st_pf == YES) pf_space_vs_time(sp, offset_sp);
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    if (msgp->rp) alg_rt_pf(sp);
    if (msgp->alg_entry == YES) alg_entry(sp, node);
#ifdef DEBUG
if (cmd.debug >= 2) debug_node("\nexpand", node, sp);
#endif
    /* expand will return a list of children */
    if ((children = expand(node, ALL_CHILDREN, DONT_CARE))) sp->stat.expanded++;
    free_node(node);

    for (child = children; child; child = child->brother) {
      sp->stat.generated++;

      if (is_infeasible(child))  {
#ifdef DEBUG
if (cmd.debug >= 3) printf("infeasible\n");
#endif
        sp->stat.infeasible++;
        free_node(child);
        continue;
      }

#ifdef DEBUG
if (cmd.debug >= 3) printf("eval bounds\n");
#endif

      evaluate_lower_bound(child);
      sol = evaluate_upper_bound(child, get_sol_buf());

#ifdef DEBUG
if (cmd.debug >= 3) debug_node(NIL, child, sp);
#endif
      bound_dom = YES;  /* init skip */

      if (child->upb < sp->glub) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("update incumbent\n");
#endif
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bounding(sp);
        dominating(child, sp);
        bound_dom = NO; /* skip over  bounded & dominated  tests */
      }
      else free_sol_buf(sol);

      if (is_feasible_or_equiv(child)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("feasible\n");
#endif
        sp->stat.feasible++;
        free_node(child);
        continue;
      }

      if (is_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
          sp->stat.bounded++;
          free_node(child);
          continue;
      }

      if (bound_dom == YES) {
        if (is_dominated(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("dominated\n");
#endif
          sp->stat.dominated++;
          free_node(child);
          continue;
        }
      }

      if (is_hard_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        sp->stat.hard_bounded++;
        free_node(child);
        continue;
      }

      /* by non-leaf, lousy-upper-bound node */
      if (bound_dom == YES) dominating(child, sp);

      list_insert(child, sp);
    } /* children loop */
  } /* selection loop */

  /* When the search is completed, the global lowb and the global upb
   * should be the same, regardless of approximation is used or not.
   * However, it is not the case, sometimes.
   * One possible reason is that
   * global lowb is supposed to be updated by a feasible sol; however,
   * the feasible sol is freed during the children loop.
   * Therefore, both are coersed to be equal.
   */
  sp->gllb = sp->glub;
  /** if (sp->approx.approx == 0.0) sp->gllb = sp->glub; **/

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);
```

```
  if (sp->constr.bound != problem.huge)
    if (sp->glub <= sp->constr.bound) msgp->feasible = YES;

  return SEARCH_IS_COMPLETED;
} /* gbb primitive */

::::::::::::::
primitive/gdfs.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ gdfs_primitive (sp, offset_sp, msgp)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
{ node_ *node, *children, *child, *p, *temp, *temp0, *next_child, *new_list;
  solution_ *sol;
  yesno_ bound_dom, to_update_gllb;
  float GUIDE_H ();
#ifdef HARE_GUIDANCE
  domain g_cost;
#endif
#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter gdfs_primitive\n");
#endif

  times(&start_time);  last_time = start_time;
  if (msgp->style == FRESH_START) {
    idpd_init();
    list_insert(create_root(sp), sp);
  }

  while ((node = list_delete(sp))) {

#ifdef HARE_GUIDANCE
    g_cost = node->g_cost;
#endif

    if (is_bounded(node, sp)) { sp->stat.bounded++; free_node(node); continue; }

    to_update_gllb = (sp->gllb == node->lowb) ? YES : NO;
    eval_rt_approx(sp);
    update_stat(sp);
    if (msgp->rt_pf == YES) pf_run_time(sp, offset_sp);
    if (msgp->st_pf == YES) pf_space_vs_time(sp, offset_sp);
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    if (msgp->rp) alg_rt_pf(sp, msgp->rp);
    if (msgp->alg_entry == YES) alg_entry(sp, node);

#ifdef DEBUG
if (cmd.debug >= 2) debug_node("\nexpand", node, sp);
#endif

    /* expand will return a list of children */
    if ((children = expand(node, ALL_CHILDREN, DONT_CARE)))
      sp->stat.expanded++;
    free_node(node);

#if defined (LOWB_GUIDANCE) || defined (UPB_GUIDANCE) || defined (UGDFS) || defined (HARE
_GUIDANCE)
#define BOUND_GUIDANCE
#endif
```

```
    new_list = NULL;
    for (child = children; child; child = next_child) {
      sp->stat.generated++;
      next_child = child->brother;

      bound_dom = YES;  /* init skip */

      if (is_infeasible(child))  {
#ifdef DEBUG
if (cmd.debug >= 3) printf("infeasible\n");
#endif
        sp->stat.infeasible++;
        free_node(child);
        continue;
      }

#ifdef DEBUG
if (cmd.debug >= 3) printf("eval bounds\n");
#endif

      evaluate_lower_bound(child);
      sol = evaluate_upper_bound(child, get_sol_buf());

#ifdef DEBUG
      if (cmd.debug >= 3) debug_node(NIL, child, sp);
#endif

      if (child->upb < sp->glub) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("update incumbent\n");
#endif
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bounding(sp);
        dominating(child, sp);
        bound_dom = NO; /* skip over  bounded & dominated  tests */
      }
      else free_sol_buf(sol);

      if (is_feasible_or_equiv(child)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("feasible\n");
#endif
        sp->stat.feasible++;
        free_node(child);
        continue;
      }

      if (is_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        sp->stat.bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES) {
        if (is_dominated(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("dominated\n");
#endif
          sp->stat.dominated++;
          free_node(child);
          continue;
        }
      }

      if (is_hard_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        if (Next_IDA_Threshold > child->lowb) Next_IDA_Threshold = child->lowb;
        sp->stat.hard_bounded++;
        free_node(child);
        continue;
      }

      /* by non-leaf, lousy-upper-bound node */
      if (bound_dom == YES) dominating(child, sp);

      child->next = new_list;
      new_list = child;

    } /* children loop */

    children = new_list;
    new_list = NULL;

#ifdef BOUND_GUIDANCE
    /** sort these live children by fancy guidance **/
    for (child = children; child; child = child->next) {
      if (new_list) {
        for (temp0 = temp = new_list; temp; temp = (temp0 = temp)->brother) {

#if defined (HARE_GUIDANCE)
          if (GUIDE_H (child->upb, child->lowb, child->depth, child->g_cost - g_cost) <
              GUIDE_H (temp->upb, temp->lowb, temp->depth, temp->g_cost - g_cost)) {
#elif defined (UPB_GUIDANCE) || defined (UGDFS)
          if (child->upb < temp->upb) {
#elif defined (LOWB_GUIDANCE)
          if (child->lowb < temp->lowb) {
#else /** bad guidance **/
          if (1) {
#endif
            if (temp == new_list) (new_list = child)->brother = temp;
            else (temp0->brother = child)->brother = temp;
            break;
          }
        }
        if (! temp) (temp0->brother = child)->brother = NULL;
      }
      else (new_list = child)->brother = NULL;
    }
    children = new_list;

#else DEFAULT_GUIDANCE
    for (child = children; child; child = child->next) {
      child->brother = new_list;
      new_list = child;
    }
    children = new_list;

#endif BOUND_GUIDANCE
```

```
    for (child = children; child; child = child->brother) {
      child->next = NULL;
      list_insert(child, sp);
    }

    if (to_update_gllb == YES) {
      if (is_list_empty(sp)) sp->gllb = sp->glub;
      else {
        /* set init value for global_lowb */
        sp->gllb = (p = sp->open.list)->lowb;
        /* global lowb == min { lowb of all active nodes } */
        for (p = p->next; p; p = p->next)
          if (p->lowb < sp->gllb) sp->gllb = p->lowb;
      }
    }

  } /* selection loop */

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

  if (sp->constr.bound != problem.huge)
    if (sp->glub <= sp->constr.bound) msgp->feasible = YES;

  /* When the search is completed, the global lowb and the global upb
   * should be the same, regardless of approximation is used or not.
   * However, it is not the case, sometimes.
   * One possible reason is that
   * global lowb is supposed to be updated by a feasible sol; however,
   * the feasible sol is freed during the children loop.
   * Therefore, both are coersed to be equal.
   */
  sp->gllb = sp->glub;
  /** if (sp->approx.approx == 0.0) sp->gllb = sp->glub; **/

  return SEARCH_IS_COMPLETED;
} /* gdfs primitive */

::::::::::::::
primitive/path.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ path_primitive (sp, offset_sp, msgp)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
{ node_ *node, *children, *child, *p, *best, *next_child;
  solution_ *sol, *best_sol;
  yesno_ bound_dom;
#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter path_primitive\n");
#endif

  times(&start_time);  last_time = start_time;
  for (node = list_delete(sp); node; node = list_delete(sp)) {
#ifdef DEBUG
if (cmd.debug >= 2) debug_node("\nexpand", node, sp);
#endif
    /* expand will return a list of children */
    children = expand(node, ALL_CHILDREN, DONT_CARE);
    free_node(node);
```

```
    best = NULL; best_sol = NULL;
    for (child = children; child; child = next_child) {
      next_child = child->brother;
      if (is_infeasible(child))  { free_node(child); continue; }
      if (child->type == ACTIVE_SINGLE) {

        evaluate_lower_bound(child);
        sol = evaluate_upper_bound(child, get_sol_buf());

        if (best) {
          if (child->lowb < best->lowb) {
            best = child;
            best_sol = sol;
            free_node (best);
          } else free_node (child);
        } else {
          best = child;
          best_sol = sol;
        }
      } else free_node (child);

    }
    sol = best_sol;
    if (! (child = best)) continue;

    if (child->upb < sp->glub) {
      sp->glub = child->upb;
      eval_rt_approx(sp);
      free_sol_buf(sp->incumbent);
      sp->incumbent = sol;
    } else free_sol_buf(sol);

    if (is_feasible_or_equiv(child)) {
      sp->stat.feasible++;
      free_node(child);
      return SEARCH_IS_COMPLETED;
    }

    list_insert(child, sp);
  } /* selection loop */

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);
  return SEARCH_IS_ABORTED;
} /* path primitive */
::::::::::::::
primitive/pbfs.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints;
 * return SEARCH_IS_IDLE, search process is idle.
 */
search_ending_ pbfs_primitive (sp, offset_sp, msgp, gllb, ngenp)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
  domain gllb;
  long *ngenp;
{ node_ *node, *children, *child;
  solution_ *sol;
  yesno_ bound_dom;
#ifdef DEBUG
```

```
if (cmd.debug >= 2) printf("\nenter pbfs_primitive\n");
#endif

  *ngenp = 0;
  times(&start_time);
  last_time = start_time;

  if ((node = btree_delete(sp))) {

    if (is_bounded(node, sp)) {
      sp->stat.bounded++;
      free_node(node);
      goto pbfs_exit;
    }

    sp->gllb = (gllb < node->lowb) ? gllb : node->lowb;
    eval_rt_approx(sp);
    update_stat(sp);
    if (msgp->rt_pf == YES) pf_run_time(sp, offset_sp);
    if (msgp->st_pf == YES) pf_space_vs_time(sp, offset_sp);
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    if (msgp->rp) alg_rt_pf(sp, msgp->rp);
    if (msgp->alg_entry == YES) alg_entry(sp, node);
#ifdef DEBUG
if (cmd.debug >= 2) debug_node("\nexpand", node, sp);
#endif
    /* expand will return a list of children */
    if ((children = expand(node, ALL_CHILDREN, DONT_CARE)))
      sp->stat.expanded++;
    free_node(node);

    for (child = children; child; child = child->brother) {
      sp->stat.generated++;
      (*ngenp)++;

      if (is_infeasible(child))  {
#ifdef DEBUG
if (cmd.debug >= 3) printf("infeasible\n");
#endif
        sp->stat.infeasible++;
        free_node(child);
        continue;
      }

#ifdef DEBUG
if (cmd.debug >= 3) printf("eval bounds\n");
#endif

      evaluate_lower_bound(child);
      sol = evaluate_upper_bound(child, get_sol_buf());

#ifdef DEBUG
if (cmd.debug >= 3) debug_node(NIL, child, sp);
#endif
      bound_dom = YES;  /* init skip */

      if (child->upb < sp->glub) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("update incumbent\n");
#endif
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bounding(sp);
        dominating(child, sp);
        bound_dom = NO; /* skip over  bounded & dominated  tests */
      }
      else free_sol_buf(sol);

      if (is_feasible_or_equiv(child)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("feasible\n");
#endif
        sp->stat.feasible++;
        free_node(child);
        continue;
      }

      if (is_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
          sp->stat.bounded++;
          free_node(child);
          continue;
      }

      if (bound_dom == YES) {
        if (is_dominated(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("dominated\n");
#endif
          sp->stat.dominated++;
          free_node(child);
          continue;
        }
      }

      if (is_hard_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        sp->stat.hard_bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES)
        dominating(child, sp);  /* by non-leaf, lousy-upper-bound node */

      btree_insert(child, sp);
    } /* children loop */
  } else {
    return SEARCH_IS_IDLE;
  }

pbfs_exit:

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

  return SEARCH_IS_COMPLETED;
} /* pbfs primitive */

::::::::::::::
primitive/pfirst.c
```

```
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints.
 */
search_ending_ pfirst_primitive (sp, offset_sp, msgp, num_active)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
  int num_active;
{ node_ *node, *children, *child, *temp;
  solution_ *sol;
  yesno_ bound_dom;
  long count;
#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter first_primitive\n");
#endif

  times(&start_time);
  last_time = start_time;
  if (msgp->style == FRESH_START) {
    idpd_init();
    btree_insert(create_root(sp), sp);
  }

  while (1) {
    for (count = 0, temp = sp->open.btree; temp; temp = temp->next)
      /** if (temp->type == ACTIVE_SINGLE) **/ count++;
    if (count >= num_active) return SEARCH_IS_COMPLETED;
    if (! (node = btree_delete(sp))) break;
    sp->gllb = node->lowb;
    update_stat(sp);
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    /* expand will return a list of children */
    if ((children = expand(node, ALL_CHILDREN, DONT_CARE))) sp->stat.expanded++;
    free_node(node);

    for (child = children; child; child = child->brother) {
      sp->stat.generated++;

      if (is_infeasible(child))  {
        sp->stat.infeasible++;
        free_node(child);
        continue;
      }

      evaluate_lower_bound(child);
      sol = evaluate_upper_bound(child, get_sol_buf());

      bound_dom = YES;  /* init skip */

      if (child->upb < sp->glub) {
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bounding(sp);
        dominating(child, sp);
        bound_dom = NO; /* skip over  bounded & dominated  tests */
      }
      else free_sol_buf(sol);

      if (is_feasible_or_equiv(child)) {
        sp->stat.feasible++;
        free_node(child);
        continue;
      }

      if (is_bounded(child, sp)) {
        sp->stat.bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES) {
        if (is_dominated(child, sp)) {
          sp->stat.dominated++;
          free_node(child);
          continue;
        }
      }

      if (is_hard_bounded(child, sp)) {
        sp->stat.hard_bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES)
        dominating(child, sp);  /* by non-leaf, lousy-upper-bound node */

      btree_insert(child, sp);
    } /* children loop */
  } /* selection loop */

  times(&break_time);
  sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
  sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

  return SEARCH_IS_COMPLETED;
} /* first primitive */
::::::::::::::
primitive/pgdfs.c
::::::::::::::

/* return SEARCH_IS_COMPLETED, search is completed;
 * return SEARCH_IS_ABORTED, search is aborted due to constraints,
 * return SEARCH_IS_IDLE, search process is idle.
 */
search_ending_ pgdfs_primitive (sp, offset_sp, msgp, gllb, ngenp)
  search_ *sp, *offset_sp;
  search_message_ *msgp;
  domain gllb;
  long *ngenp;
{ node_ *node, *children, *child, *p, *temp, *temp0, *next_child, *new_list;
  solution_ *sol;
  yesno_ bound_dom;
#ifdef DEBUG
if (cmd.debug >= 2) printf("\nenter pgdfs_primitive\n");
#endif

  *ngenp = 0;
  times(&start_time);
  last_time = start_time;

  if ((node = list_delete(sp))) {

    if (is_bounded(node, sp)) {
      sp->stat.bounded++;
      free_node(node);
      goto pgdfs_exit;
    }

    sp->gllb = (gllb < node->lowb) ? gllb : node->lowb;
    eval_rt_approx(sp);
    update_stat(sp);
    if (msgp->rt_pf == YES) pf_run_time(sp, offset_sp);
    if (msgp->st_pf == YES) pf_space_vs_time(sp, offset_sp);
    if (is_constr_violated(sp)) return SEARCH_IS_ABORTED;
    if (msgp->rp) alg_rt_pf(sp, msgp->rp);
    if (msgp->alg_entry == YES) alg_entry(sp, node);
#ifdef DEBUG
if (cmd.debug >= 2) debug_node("\nexpand", node, sp);
#endif
    /* expand will return a list of children */
    if ((children = expand(node, ALL_CHILDREN, DONT_CARE)))
      sp->stat.expanded++;
    free_node(node);

#if defined (LOWB_GUIDANCE) || defined (UPB_GUIDANCE) || defined (UGDFS)
#define BOUND_GUIDANCE
#endif

    new_list = NULL;
    for (child = children; child; child = next_child) {
      sp->stat.generated++;
      (*ngenp)++;
      next_child = child->brother;

      bound_dom = YES;  /** init skip **/

      if (is_infeasible(child))  {
#ifdef DEBUG
if (cmd.debug >= 3) printf("infeasible\n");
#endif
        sp->stat.infeasible++;
        free_node(child);
        continue;
      }

#ifdef DEBUG
if (cmd.debug >= 3) printf("eval bounds\n");
#endif

      evaluate_lower_bound(child);
      sol = evaluate_upper_bound(child, get_sol_buf());

#ifdef DEBUG
if (cmd.debug >= 3) debug_node(NIL, child, sp);
#endif

      if (child->upb < sp->glub) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("update incumbent\n");
#endif
        sp->glub = child->upb;
        eval_rt_approx(sp);
        free_sol_buf(sp->incumbent);
        sp->incumbent = sol;
        bounding(sp);
        dominating(child, sp);
        bound_dom = NO; /* skip over  bounded & dominated  tests */
      }
```

```
      else free_sol_buf(sol);

      if (is_feasible_or_equiv(child)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("feasible\n");
#endif
        sp->stat.feasible++;
        free_node(child);
        continue;
      }

      if (is_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        sp->stat.bounded++;
        free_node(child);
        continue;
      }

      if (bound_dom == YES) {
        if (is_dominated(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("dominated\n");
#endif
          sp->stat.dominated++;
          free_node(child);
          continue;
        }
      }

      if (is_hard_bounded(child, sp)) {
#ifdef DEBUG
if (cmd.debug >= 3) printf("bounded\n");
#endif
        sp->stat.hard_bounded++;
        free_node(child);
        continue;
      }

      /* by non-leaf, lousy-upper-bound node */
      if (bound_dom == YES) dominating(child, sp);

      child->next = new_list;
      new_list = child;

    } /* children loop */

    children = new_list;
    new_list = NULL;

#ifdef BOUND_GUIDANCE
    /** sort these live children by fancy guidance **/
    for (child = children; child; child = child->next) {
      if (new_list) {
        for (temp0 = temp = new_list; temp; temp = (temp0 = temp)->brother) {

#if defined (UPB_GUIDANCE) || defined (UGDFS)
          if (child->upb < temp->upb) {
#elif defined (LOWB_GUIDANCE)
          if (child->lowb < temp->lowb) {
#else /** bad guidance **/
          if (1) {
#endif
```

```
      if (temp == new_list) (new_list = child)->brother = temp;
      else (temp0->brother = child)->brother = temp;
      break;
     }
    }
    if (! temp) (temp0->brother = child)->brother = NULL;
   }
   else (new_list = child)->brother = NULL;
  }
  children = new_list;

#else DEFAULT_GUIDANCE
  for (child = children; child; child = child->next) {
   child->brother = new_list;
   new_list = child;
  }
  children = new_list;

#endif BOUND_GUIDANCE

  for (child = children; child; child = child->brother) {
   child->next = NULL;
   list_insert(child, sp);
  }

 } else {

  return SEARCH_IS_IDLE;
 }

pgdfs_exit:

 times(&break_time);
 sp->stat.unix_utime += (break_time.tms_utime - start_time.tms_utime);
 sp->stat.unix_stime += (break_time.tms_stime - start_time.tms_stime);

 if (sp->constr.bound != problem.huge)
  if (sp->glub <= sp->constr.bound) msgp->feasible = YES;

 return SEARCH_IS_COMPLETED;
} /* pgdfs primitive */
```

```
Fri Jan 31 10:00:10 CST 1992
::::::::::::::
kernel/etc.c
::::::::::::::

void error (msg)
  char *msg;
{ fprintf(stderr, "*** ERROR ***  %s", msg); exit(0); }


int ceiling (val)
  float val;
{ int temp;

  temp = toint(val);
  if (val > tofloat(temp)) temp++;
  return temp;
}


int is_any (array, left, right)
  int array[], left, right;
{ int k;

  for (k = left; k <= right; k++)
    if (array[k]) return 1;
  return 0;
}


int is_member (item, array, left, right)
  int item, array[], left, right;
{ int i;

  for (i = left; i < right; i++)
    if (item == array[i]) return 1;
  return 0;
}


int is_invalid_gllb (sp)
  search_ *sp;
{ return ((tofloat(sp->gllb) == 0.0) || (sp->gllb == - problem.huge)); }


int is_compound (n)
  node_ *n;
{ return (n->type == ACTIVE_COMPOUND); }


void set_compound (n)
  node_ *n;
{ n->type = ACTIVE_COMPOUND; }


/* internal maintenance of solution buffers */
solution_       *sol_buf_mgr = NULL;

solution_ *get_sol_buf ()
{ solution_ *p;

  if (sol_buf_mgr) {
    p = sol_buf_mgr;
    sol_buf_mgr = ((solution_ *) *((int *) p));
  }
  else {
    p = (solution_ *) malloc(2 * sizeof(solution_));
    sol_buf_mgr = p + 1;
    *((int *) sol_buf_mgr) = NULL;
  }
  reset_sol_buf(p);
  return p;
}


void free_sol_buf (solp)
  solution_ *solp;
{
  *((int *) solp) = ((int) sol_buf_mgr);
  sol_buf_mgr = solp;
}


int is_alg_rt_pf (sp)
  search_ *sp;
{
  switch(sp->algorithm) {
    case pTCA:
    case dTCA: return 1;
    default:   return 0;
  }
}


/* generate a random integer over [left,right] */
int gen_random_int (left, right)
  int left, right;
{ return (left + ((int) (gen_random_float() * (right - left + 1)))); }


/* generate a random float over [0.0,1.0) */
float gen_random_float ()
{ return (((float) rand()) / RANDOM_RADIX); }


/* generate a random float over [left,right) */
float gen_random_range (left, right)
  float left, right;
{ return (left + gen_random_float() * (right - left)); }


void set_search_task (task)
  task_ task;
{ problem.task = task; }


void debug_node (message, node, sp)
  char *message;
  node_ *node;
  search_ *sp;
{
  if (message) printf("%s\n", message);

  if (node) {
    printf("node=%x, parent=%x, depth=%d, entity=%d, ",
                node, node->parent, node->depth, node->entity);
    if (problem.domain == INT || problem.domain == LONG)
      printf("g_cost=%d, lowb=%d, upb=%d, glub=%d, gllb=%d\n",
            node->g_cost, node->lowb, node->upb, sp->glub, sp->gllb);
    else
      printf("g_cost=%g, lowb=%g, upb=%g, glub=%g, gllb=%g\n",
             node->g_cost, node->lowb, node->upb, sp->glub, sp->gllb);
  }
}


void set_node_size (pdsd_size)
  long pdsd_size;
{ node_conf.node_size = (node_conf.pdsd_size = pdsd_size) + sizeof(node_); }


void junk() { printf("checkpoint\n"); }

::::::::::::::
kernel/free.c
::::::::::::::

void free_environ (sp, not_top_level)
  search_ *sp;
  int not_top_level;
{ search_ *p;

  if (sp) {

    if (sp->child) free_environ (sp->child, 1);
    if (sp->brother) free_environ (sp->brother, 1);

    free_open(sp);

    if (not_top_level) {
      free_alg_struct(sp);
      free(sp);
    }
  }
}


void free_alg_struct (sp)
  search_ *sp;
{
  if (sp->alg.def)
    switch (sp->algorithm)
    {
      case DEFAULT:                         sp->alg.def = NULL;   break;
      case LW:                              sp->alg.lw = NULL;    break;
      case sTCA:  free (sp->alg.stca);  sp->alg.stca = NULL;  break;
      case pTCA:  free (sp->alg.ptca);  sp->alg.ptca = NULL;  break;
      case dTCA:  free (sp->alg.dtca);  sp->alg.dtca = NULL;  break;
      case sTCGD: free (sp->alg.stcgd); sp->alg.stcgd = NULL; break;
      case pTCGD: free (sp->alg.ptcgd); sp->alg.ptcgd = NULL; break;
      case uRTS:  free (sp->alg.urts);  sp->alg.urts = NULL;  break;
      case bRTS:  free (sp->alg.brts);  sp->alg.brts = NULL;  break;
      case hRTS:  free (sp->alg.hrts);  sp->alg.hrts = NULL;  break;
      case uIRD:  free (sp->alg.uird);  sp->alg.uird = NULL;  break;
      case bIRD:  free (sp->alg.bird);  sp->alg.bird = NULL;  break;
      default:    error ("free_alg_struct: no such algorithm\n"); break;
    }
}


void free_open (sp)
  search_ *sp;
{ int i;

  switch (sp->strategy)
  {
    case GBB:  free_list(sp->open.list);   sp->open.list = NULL;  break;
    case BFS:  free_bptree(sp->open.bptree); sp->open.bptree = NULL; break;
    case DFS:  free_stack(sp->open.stack); sp->open.stack = NULL; break;
    case GDFS: free_list(sp->open.list);   sp->open.list = NULL;  break;
    case BAND: free_btree(sp->open.btree); sp->open.btree = NULL;
               for (i = 0; i < MaxDepth; i++) {
                 free_btree ((sp->band.tree+i)->young);
                 (sp->band.tree+i)->young = NULL;
               }
               break;
    default:   free_list(sp->open.list);   sp->open.list = NULL;  break;
  }
}


void free_list (list)
  node_ *list;
{ node_ *p, *p0;

  for (p = list; p; p = p0) {
    p0 = p->next;
    free_node(p);
  }
}


void free_btree (tree)
  node_ *tree;
{ node_ *p, *p0;

  for (p = tree; p; p = p0) {
    p0 = p->next;
    free_node(p);
  }
}


void free_stack (stk)
  node_ *stk;
{ node_ *p, *p0;

  for (p = stk; p; p = p0) {
    p0 = p->next;
    free_node(p);
  }
}


#ifndef DELAY_FREE_NODE
void free_node (node)
  node_ *node;
{ node_ *parentp;

  /* bad-node test */
  if (node == NULL) return;

  /* reset activeness of node */
  node->type = INACTIVE;

  /* free this node */
  dispose_search_node_to_pool(node);
}

#else

void free_node (node)
  node_ *node;
{ node_ *parentp;

  /* bad-node test */
  if (node == NULL) return;

  /* reset activeness of node */
  node->type = INACTIVE;

  if (node->nsprout <= 0) {
    while (node) {

      /* remember parent */
      parentp = node->parent;

#ifdef DEBUG
if (cmd.debug >= 3) printf("free node=%d\n", node);
#endif

      /* free this node */
      dispose_search_node_to_pool(node);

      /* update parent's nsprout
       * further if it is zero, then free parent also
       */
      if (parentp)
        if (--(parentp->nsprout) > 0) node = NULL;
        else node = parentp;
      else node = NULL;
    }
  }
#ifdef DEBUG
  else
    if (cmd.debug >= 3)
      printf("not pruned due to nsprout=%d\n", node->nsprout);
#endif
}
#endif

::::::::::::::
kernel/init.c
::::::::::::::

search_ *init_environ (sp)
  search_ *sp;
{ search_ *new_sp;
  domain base;

  new_sp = (sp) ? sp : ((search_ *) malloc(sizeof(search_)));

  if (sp) {
    init_search_struct(new_sp, 1, 0);
    init_alg_struct(new_sp);
  }
  else {
    init_search_struct(new_sp, 1, 1);
    attach_alg_struct(new_sp);
    clear_files();
  }

  idpd_init();
  examine_root(new_sp);
  if (cmd.bound_metric == RELATIVE) {
    base = (cmd.bound_base == problem.huge) ? ROOT->upb : cmd.bound_base;
    new_sp->constr.bound = ROOT->lowb +
                 (base - ROOT->lowb) * cmd.constr.bound / 100.0;
  }
  return new_sp;
}


void clear_files ()
{ char cmd[100];

  fclose(fopen(io.stat,"w"));
  fclose(fopen(io.summary,"w"));
  fclose(fopen(io.rt_pf,"w"));
  fclose(fopen(io.st_pf,"w"));
  fclose(fopen(io.report,"w"));

  sprintf(cmd, "rm %s %s %s %s %s",
               io.stat, io.summary, io.rt_pf, io.st_pf, io.report);
  system(cmd);
}


void attach_alg_struct (sp)
  search_ *sp;
{
  switch (sp->algorithm)
  {
    case sTCA:  sp->alg.stca = (stca_ *) malloc (sizeof (stca_));     break;
    case pTCA:  sp->alg.ptca = (ptca_ *) malloc (sizeof (ptca_));     break;
    case dTCA:  sp->alg.dtca = (dtca_ *) malloc (sizeof (dtca_));     break;
    case sTCGD: sp->alg.stcgd = (stcgd_ *) malloc (sizeof (stcgd_));  break;
    case pTCGD: sp->alg.ptcgd = (ptcgd_ *) malloc (sizeof (ptcgd_));  break;
    case uRTS:  sp->alg.urts = (urts_ *) malloc (sizeof (urts_));     break;
    case bRTS:  sp->alg.brts = (brts_ *) malloc (sizeof (brts_));     break;
    case hRTS:  sp->alg.hrts = (hrts_ *) malloc (sizeof (hrts_));     break;
    case uIRD:  sp->alg.uird = (uird_ *) malloc (sizeof (uird_));     break;
    case bIRD:  sp->alg.bird = (bird_ *) malloc (sizeof (bird_));     break;
    default:     break;
  }
  init_alg_struct (sp);
}


void init_alg_struct (sp)
  search_ *sp;
{
  switch (sp->algorithm)
  {
    case sTCA:  init_stca_struct (sp->alg.stca);       break;
    case pTCA:  init_ptca_struct (sp->alg.ptca);       break;
    case dTCA:  init_dtca_struct (sp->alg.dtca);       break;
    case sTCGD: init_stcgd_struct (sp->alg.stcgd);     break;
    case pTCGD: init_ptcgd_struct (sp->alg.ptcgd);     break;
    case uRTS:  init_urts_struct (sp->alg.urts);       break;
    case bRTS:  init_brts_struct (sp->alg.brts);       break;
    case hRTS:  init_hrts_struct (sp->alg.hrts);       break;
    case uIRD:  init_uird_struct (sp->alg.uird);       break;
    case bIRD:  init_bird_struct (sp->alg.bird);       break;
    default:    break;
  }
}


void examine_root (sp)
  search_ *sp;
{ node_ *root;
  solution_ *solp;

  /* set init threshold */
  ROOT = root = root_generator();
  evaluate_lower_bound(root);
  solp = evaluate_upper_bound(root, get_sol_buf());
  if (sp->incumbent == NULL) sp->incumbent = solp;
  else free_sol_buf(solp);
  sp->approx.x_root_approx = sp->approx.root_approx
                 = calc_rt_approx(root->lowb, root->upb, NO, problem.huge);
  sp->gllb = sp->x_root_lb = sp->root_lb = root->lowb;
  sp->glub = sp->x_root_ub = sp->root_ub = root->upb;
  sp->approx.root_accu = approx2accu (sp->approx.root_approx);

#ifdef DEBUG
if (cmd.debug >= 1)
  printf("root: lowb=%g, upb=%g, root_approx=%g\n",
        ((float) sp->root_lb), ((float) sp->root_ub), sp->approx.root_approx);
#endif
}


node_ *create_root (sp)
  search_ *sp;
{ node_ *root;
  solution_ *solp;

#ifdef DEBUG
if (cmd.debug >= 2) printf("generate root\n");
#endif

  root = root_generator();

  evaluate_lower_bound(root);
  solp = evaluate_upper_bound(root, get_sol_buf());

  sp->root_lb = root->lowb;
  sp->root_ub = root->upb;

  if (sp->incumbent == NULL) {
    sp->incumbent = solp;
    sp->gllb = sp->x_root_lb = root->lowb;
    sp->glub = sp->x_root_ub = root->upb;
  }
  else free_sol_buf (solp);

#ifdef DEBUG
if (cmd.debug >= 2) debug_node("root", root, sp);
#endif

  return root;
}

::::::::::::::
kernel/limit.c
::::::::::::::

yesno_ is_null_device (sp)
  search_ *sp;
{
  if (sp->approx.approx > 0.0) return NO;
  if (sp->constr.bound < problem.huge) return NO;

  return YES;
}


domain expected_opt (sp)
  search_ *sp;
{ float temp;

  temp = tofloat(sp->glub) / (1.0 + sp->approx.approx);
  switch (problem.domain) {
    case INT:
    case LONG:   return ((domain) ceiling(temp)); break;
    case FLOAT:
    case DOUBLE:
    default:     break;
  }
  return ((domain) temp);
}


float calc_rt_approx (lowb, upb, th_flag, th)
  domain lowb, upb;
  yesno_ th_flag;
  domain th;
{ float rt, rt_th, up = tofloat(upb), low = tofloat(lowb);

  /* normal run-time approx */
  rt = (low == ((domain) 0) || lowb == problem.huge)
                                             ? HUGE_FLOAT : (up - low) / low;
  rt = max(rt,0.0);

  /* threshold-induced run-time approx */
  if (th_flag == YES) {
    if (th == problem.huge) rt_th = 0.0;
    else rt_th = (th == ((domain) 0)) ? HUGE_FLOAT : (up - th) / th;
    rt = max(rt,rt_th);
  }

  return rt;
}


float get_rt_approx (sp)
  search_ *sp;
{ domain th = problem.huge;
  yesno_ th_flag = NO;

  if (is_threshold_related(sp) == YES) {
    th = get_threshold(sp);
    th_flag = YES;
  }
  return calc_rt_approx(sp->gllb, sp->glub, th_flag, th);
}


void eval_rt_approx(sp)
  search_ *sp;
```

```
{ domain th = problem.huge;
  yesno_ th_flag = NO;

  if (is_threshold_related(sp) == YES) {
    th = get_threshold(sp);
    th_flag = YES;
  }
  sp->approx.run_time = calc_rt_approx(sp->gllb, sp->glub, th_flag, th);
}


float get_approx (sp)
  search_ *sp;
{ return (max(sp->approx.run_time,sp->approx.approx)); }


void put_achieved_approx (sp, a)
  search_ *sp;
  float a;
{
  if (sp->approx.achieved > a) sp->approx.achieved = a;
}


void eval_final_approx (sp)
  search_ *sp;
{
  eval_rt_approx(sp);
  put_achieved_approx(sp, get_approx(sp));
}


yesno_ is_threshold_related (sp)
  search_ *sp;
{
  switch (sp->algorithm) {
    case uRTS: return ((sp->alg.urts->alg == UNARY_TH) ? YES : NO); break;
    case bRTS: return YES; break;
    case hRTS: return YES; break;
    case uIRD: return ((sp->alg.uird->alg == UNARY_TH) ? YES : NO); break;
    case bIRD: return YES; break;
    case IDA:  return YES; break;
    case DFS_star:  return YES; break;
    default:     break;
  }
  return NO;
}


domain get_threshold (sp)
  search_ *sp;
{
  switch (sp->algorithm) {
    case uRTS:  return sp->alg.urts->th;        break;
    case bRTS:  return sp->alg.brts->th;        break;
    case hRTS:  return sp->alg.hrts->th;        break;
    case uIRD:  return sp->alg.uird->th;        break;
    case bIRD:  return sp->alg.bird->th;        break;
    case IDA:   return sp->constr.bound;        break;
    case DFS_star: return sp->constr.bound;     break;
    default:      error("get_threshold: no such algorithm"); break;
  }
  return problem.huge;
}
```

```
void put_threshold (sp, new_th)
  search_ *sp;
  domain new_th;
{
  switch (sp->algorithm) {
    case uRTS:  sp->alg.urts->th = new_th;      break;
    case bRTS:  sp->alg.brts->th = new_th;      break;
    case hRTS:  sp->alg.hrts->th = new_th;      break;
    case uIRD:  sp->alg.uird->th = new_th;      break;
    case bIRD:  sp->alg.bird->th = new_th;      break;
    case IDA:   sp->constr.bound = new_th;      break;
    case DFS_star:   sp->constr.bound = new_th; break;
    default:    error("put_threshold: no such algorithm"); break;
  }
}


void merge_threshold_to_parent (child_sp, sp)
  search_ *child_sp, *sp;
{ domain th = get_threshold(child_sp);

  put_threshold(sp, th);
  /** sp->gllb = th; **/
}


/* return 1, if violated;
 * return 0, otherwise.
 */
int is_constr_violated (sp)
  search_ *sp;
{ int signal = 0;

  if ( (get_time(sp) >= sp->constr.time)    ||
       (get_space(sp) >= sp->constr.space)  ||
       (get_cst(sp) >= sp->constr.cst)         )  signal = 1;

#ifdef DEBUG
if (cmd.debug >= 2)
  if (signal) {
    printf("constr is violated: ");
    if (get_time(sp) >= sp->constr.time) printf("time(YES), ");
    else printf("time(NO), ");
    if (get_space(sp) >= sp->constr.space) printf("space(YES), ");
    else printf("space(NO), ");
    if (get_cst(sp) >= sp->constr.cst) printf("cst(YES)\n");
    else printf("cst(NO)\n");
    printf("(%d,%d), (%d,%d), (%f,%f)\n", get_time(sp), sp->constr.time,
           get_space(sp), sp->constr.space, get_cst(sp), sp->constr.cst);
  }
  else printf("no constr is violated\n");
#endif

  return signal;
}


void put_child_constr (childp, parentp, t_percent, s_percent, cst_percent)
  search_ *childp, *parentp;
  float t_percent, s_percent, cst_percent;
{ float time, space;
```

```
  if (parentp->constr.time == huge_long) childp->constr.time = HUGE_LONG;
  else {
    time = (float) (parentp->constr.time - get_time(parentp));
    time = time * t_percent;
    childp->constr.time = tolong(time);
  }
  if (parentp->constr.space == huge_long) childp->constr.space = HUGE_LONG;
  else {
    space = (float) (parentp->constr.space - get_space(parentp));
    space = space * s_percent;
    childp->constr.space = tolong(space);
  }
  childp->constr.cst = (parentp->constr.cst == huge_float) ?
          HUGE_FLOAT : (parentp->constr.cst - get_cst(parentp)) * cst_percent;
}


void set_stop_constr (sp, t_percent, s_percent, cst_percent)
  search_ *sp;
  float t_percent, s_percent, cst_percent;
{ float time = tofloat(sp->constr.time), space = tofloat(sp->constr.space);

  time = time * t_percent;    sp->constr.time = tolong(time);
  space = space * s_percent;  sp->constr.space = tolong(space);
  sp->constr.cst = sp->constr.cst * cst_percent;
}

::::::::::::::
kernel/main.c
::::::::::::::

main (argc, argv)
  int argc;
  char *argv[];
{ int iter;
  search_ *top_sp;

  /* init some data structures */
  init_problem_struct(&problem);
  init_cmd_struct(&cmd);
  init_io_struct(&io);

  pool_manager = NULL;

  /* parse the command line */
  cmd_line(argc, argv);

  /* problem dependent initialization */
  iipd_init();

  /* start testbed execution wiht user-specified number of iterations */
  for (top_sp = NULL, iter = 1; iter <= cmd.iter; iter++) {

    /* feed the user-supplied seed to the random generator */
    srand(problem.seed);

    /* generate a sample problem for this iteration */
    gen_sample_problem();

    /* initialize environment for current iteration */
    top_sp = init_environ(top_sp);

    /* start search operation */
    search(top_sp);

    /* flush out final result */
    flush(top_sp);

#ifdef DEBUG
if (cmd.debug >= 2) evaluate_solution(top_sp);
#endif DEBUG

    /* keep the result of this iteration */
    if (problem.task == LEARNING) keep_result(iter, top_sp);

    /* free the environment */
    free_environ(top_sp, 0 /* not_top_level = 0 */);

    /* inc rand seed by one */
    problem.seed++;
  } /* iter loop */

  /* print out all the results to the output file named by argv[4] */
  if (problem.task == LEARNING) print_result(--iter, io.stat);
} /* main */

::::::::::::::
kernel/para.c
::::::::::::::
yesno_ para_termination (pe, num_pe)
  search_ *pe[];
  int num_pe;
{ int i;

  for (i = 0; i < num_pe; i++)
    if (pe[i]->open.list) return NO;
  return YES;
}


void para_merge_solution (sp, pe, num_pe)
  search_ *sp;
  search_ *pe[];
  int num_pe;
{ int i;
  domain gllb, glub;
  solution_ *incumbent;

  gllb = pe[0]->gllb;
  glub = pe[0]->glub;
  incumbent = pe[0]->incumbent;
  for (i = 1; i < num_pe; i++)
  {
    if (gllb > pe[i]->gllb) gllb = pe[i]->gllb;
    if (glub > pe[i]->glub)
    {
      glub = pe[i]->glub;
      incumbent = pe[i]->incumbent;
    }
  }
  sp->approx.run_time = ((float) (glub - gllb)) / tofloat (gllb);
  sp->gllb = gllb;
  sp->glub = glub;
}


void para_merge_stat (sp, pe, num_pe)
  search_ *sp;
  search_ *pe[];
  int num_pe;
{ int i;
  stat_ *p = &(sp->stat), *c;

  for (i = 0; i < num_pe; i++)
  {
    c = &(pe[i]->stat);
    p->generated += c->generated;
    p->expanded += c->expanded;
    p->feasible += c->feasible;
    p->infeasible += c->infeasible;
    p->bounded += c->bounded;
    p->dominated += c->dominated;
    p->bounding += c->bounding;
    p->dominating += c->dominating;
    p->hard_bounded += c->hard_bounded;
    p->time += c->time;
    p->v_cst += c->v_cst;
    p->r_cst += c->r_cst;
    p->unix_utime += c->unix_utime;
    p->unix_stime += c->unix_stime;
  }

  p->active = pe[0]->stat.active;
  p->max_active = pe[0]->stat.max_active;
  for (i = 1; i < num_pe; i++)
  {
    c = &(pe[i]->stat);
    if (c->active > p->active) p->active = c->active;
    if (c->max_active > p->max_active) p->max_active = c->max_active;
  }
}


void para_load_balancing (pe, num_pe)
  search_ *pe[];
  int num_pe;
{ int i;
  node_ *p;

  for (i = 0; i < num_pe; i++)
  {
    if (pe[i]->open.list == NULL)
      if ((p = para_request (pe, i, num_pe))) insert (p, pe[i]);
  }
}


node_ *para_request (pe, who, num_pe)
  search_ *pe[];
  int who, num_pe;
{ int i;

  if (pe[who]->idle_clk >= cmd.comm_idle)
  {
    for (i = 0; i < num_pe; i++)
      if (pe[i]->stat.active > 1)
      {
        pe[who]->idle_clk = 0;
        return (delete (pe[i]));
      }
  } else (pe[who]->idle_clk)++;
  return NULL;
```

```
}

::::::::::::::
kernel/profile.c
::::::::::::::

#define TRANSIENT_PHASE         20

int is_in_transient_phase (sp)
  search_ *sp;
{ return((get_time(sp) < TRANSIENT_PHASE)); }


/* if we are doing run-time profile, Real/Virtual_Time_Limit is used as
 * step size for recording run-time approximation
 */
void pf_run_time (sp, offset_sp)
  search_ *sp, *offset_sp;
{ FILE *fp;
  float accu;

  if (is_in_transient_phase(sp)) return;
  eval_rt_approx(sp);

  if (cmd.xon == YES) {
    accu = approx2accu (sp->approx.run_time);
    if (cmd.pf.last_accu - accu >= cmd.pf.step) {
      cmd.pf.last_accu = accu;
      if (accu != 0.0) {
        fp = fopen(io.rt_pf, "a");
        accu = get_offset_rt_approx(sp, offset_sp);
        accu = approx2accu (accu);
        fprintf(fp, "%g  %d  %g  %d\n",
              accu,
              get_offset_time(sp, offset_sp),
              get_offset_cst(sp, offset_sp),
              get_offset_gen(sp, offset_sp));
        fclose(fp);
      }
    }
  } else {
    if ((cmd.pf.last_approx - sp->approx.run_time) >= cmd.pf.step) {
      cmd.pf.last_approx = sp->approx.run_time;
      if (sp->approx.run_time != huge_float) {
        fp = fopen(io.rt_pf, "a");
        fprintf(fp, "%g  %d  %g  %d\n",
              get_offset_rt_approx(sp, offset_sp),
              get_offset_time(sp, offset_sp),
              get_offset_cst(sp, offset_sp),
              get_offset_gen(sp, offset_sp));
        fclose(fp);
      }
    }
  }
}


void pf_space_vs_time (sp, offset_sp)
  search_ *sp, *offset_sp;
{ FILE *fp;

  fp = fopen(io.st_pf, "a");
  fprintf(fp, "%d  %d  %g  %d\n",
        get_offset_time(sp, offset_sp),  get_offset_space(sp, offset_sp),
          get_offset_cst(sp, offset_sp),   get_offset_gen(sp, offset_sp));
  fclose(fp);
}


/* collect run-time profile for some search algorithms */
void alg_rt_pf (sp, rp)
  search_ *sp;
  regress_ *rp;
{ long time = get_time(sp);
  float a = sp->approx.run_time;
  double x, y;

  if (cmd.xon == YES) {
    a = approx2accu (a);
    if (time == 0 || a == 0.0) return;
  } else {
    if (time == 0 || a == huge_float) return;
  }
  x = log10(todouble(time));
  y = todouble(a);
  rp->n += 1.0;
  rp->x_sum += x;
  rp->x2_sum += (x * x);
  rp->y_sum += y;
  rp->xy_sum += (x * y);
}


/* collect execution profile (search by search) for some search algorithms */
void alg_pf (sp, rp)
  search_ *sp;
  regress_ *rp;
{ long time = get_time(sp);
  float a = get_approx(sp);
  double x, y;

  if (cmd.xon == YES) {
    a = approx2accu (a);
    if (time == 0 || a == 0.0) return;
  } else {
    if (time == 0 || a == huge_float) return;
  }
  x = log10(todouble(time));  y = todouble(a);
#ifdef DEBUG
if (cmd.debug >= 1) printf("doing alg pf over searches\n");
#endif
  rp->n += 1.0;
  rp->x_sum += x;
  rp->x2_sum += (x * x);
  rp->y_sum += y;
  rp->xy_sum += (x * y);
}


void alg_pf_predict (sp, rp)
  search_ *sp;
  regress_ *rp;
{ double det;
  float tmp;

  det = rp->n * rp->x2_sum - rp->x_sum * rp->x_sum;
  rp->beta[0] = (rp->x2_sum * rp->y_sum - rp->x_sum * rp->xy_sum) / det;
  rp->beta[1] = (- rp->x_sum * rp->y_sum + rp->n * rp->xy_sum) / det;
```

```
  tmp = (float) (rp->beta[0] + rp->beta[1] * log10(todouble(sp->constr.time)));
#ifdef DEBUG
if (cmd.debug >= 1) {
  printf("n=%f, x2_sum=%f, x_sum=%f\n", rp->n, rp->x2_sum, rp->x_sum);
  printf("det=%f, beta0=%f, beta1=%f\n", det, rp->beta[0], rp->beta[1]);
  printf("old pred=%g,  new pred=%f\n", sp->approx.predicted, tofloat(tmp));
}
#endif

  if (cmd.xon == YES) tmp = accu2approx (tmp);
  sp->approx.predicted = tmp;
  if (sp->approx.predicted < 0.0) sp->approx.predicted = 0.0;
}

::::::::::::::
kernel/search.c
::::::::::::::

#define GetPE   ((search_ **) malloc (cmd.num_pe * sizeof (search_ *)))

/* the search starter per iteration,
 * the stat structure will be initialized, since it is entirely a new search.
 */
void search (sp)
  search_ *sp;
{
  switch (sp->algorithm)
  {
    case DEFAULT: default_algorithm (sp);                  break;
    case LW:    lawler_wood_algorithm (sp);                break;
    case sTCA:  stca_algorithm (sp);                       break;
    case pTCA:  ptca_algorithm (sp);                       break;
    case dTCA:  dtca_algorithm (sp);                       break;
    case sTCGD: stcgd_algorithm (sp);                      break;
    case pTCGD: ptcgd_algorithm (sp);                      break;
    case uRTS:  urts_algorithm (sp);                       break;
    case bRTS:  brts_algorithm (sp);                       break;
    case hRTS:  hrts_algorithm (sp);                       break;
    case uIRD:  uird_algorithm (sp);                       break;
    case bIRD:  bird_algorithm (sp);                       break;
    case IRA:   ira_algorithm (sp);                        break;
    case IDA:   ida_algorithm (sp);                        break;
    case DFS_star:      ida_algorithm (sp);                break;
    case PBFS:  para_algorithm (sp, GetPE, cmd.num_pe); break;
    case PGDFS: para_algorithm (sp, GetPE, cmd.num_pe); break;
    case PBAND: para_algorithm (sp, GetPE, cmd.num_pe); break;
    default:    error ("search: no such algorithm\n");  break;
  }
}
#undef GetPE


search_ *create_brother_search (sp)
  search_ *sp;
{ search_ *new_sp;

  new_sp = (search_ *) malloc(sizeof(search_));
  init_search_struct(new_sp, 1, 1);
  new_sp->strategy = sp->strategy;
  new_sp->algorithm = sp->algorithm;

  new_sp->brother = sp;
  new_sp->parent = sp->parent;
  attach_alg_struct(new_sp);
  return new_sp;
}


search_ *create_child_search (sp)
  search_ *sp;
{ search_ *new_sp;

  new_sp = (search_ *) malloc(sizeof(search_));
  init_search_struct(new_sp, 1, 1);
  new_sp->strategy = sp->strategy;
  new_sp->algorithm = sp->algorithm;

  new_sp->brother = sp->child;
  new_sp->parent = sp;
  sp->child = new_sp;

  attach_alg_struct(new_sp);
  return new_sp;
}


/* some book-keeping for certain algorithms */
void alg_entry (sp, node)
  search_ *sp;
  node_ *node;
{
  switch (sp->algorithm) {
    case dTCA:  dtca_entry(sp);         break;
    case uRTS:  urts_entry(sp, node);   break;
    case bRTS:  brts_entry(sp, node);   break;
    case hRTS:  hrts_entry(sp, node);   break;
    default:                                    break;
  }
}

#define is_alg_skip_pruning     (RTS_IRD_iter == 1 && sp->stat.feasible <= 0)

int is_bounded (child, sp)
  node_ *child;
  search_ *sp;
{
#if defined (RTS_ENHANCE) || defined (IRD_ENHANCE)
  if (is_alg_skip_pruning) return 0;
#endif
  return (child->lowb >= expected_opt(sp) ? 1 : 0);
}


int is_hard_bounded (child, sp)
  node_ *child;
  search_ *sp;
{
#if defined (RTS_ENHANCE) || defined (IRD_ENHANCE)
  if (is_alg_skip_pruning) return 0;
#endif
  if (No_Upper_Bound == YES) return (child->lowb > sp->constr.bound ? 1 : 0);
  else return (child->lowb >= sp->constr.bound ? 1 : 0);
}

#undef is_alg_skip_pruning
```

```
void dominating (node, sp) node_ *node; search_ *sp; { }


void bounding (sp)
  search_ *sp;
{ int i;
  void young_bounding ();

  switch (sp->strategy) {
    case BFS:  bptree_bounding(sp); break;
    case GDFS: list_bounding(sp);  break;
    case BAND: btree_bounding(sp);
               for (i = sp->band.ceiling_depth; i <= sp->band.floor_depth; i++)
                  young_bounding (sp, i);
               break;
    default:                        break;
  }
}



void bptree_bounding (sp)
  search_ *sp;
{
#ifdef BPLUS_TREE
  node_ *p, *q;
  long free_bptree ();

  for (p = sp->open.bptree; p; ) {
    if (is_bounded (p, sp)) {
      if (p == sp->open.bptree) sp->open.bptree = p->left;
      if (p->up) p->up->right = p->left;
      if (p->left) p->left->up = p->up;

      /* re-check its left subtree */
      q = p;
      p = p->left;

      /* rm itself and its right subtree */
      q->left = NULL;
      sp->stat.active -= free_bptree (q);

    } else p = p->right;
  }
#else
  btree_bounding (sp);
#endif
}


long free_bptree (p)
  node_ *p;
{ long count = 0;

  if (p) {
    count = 1;
    if (p->left)  count += free_bptree (p->left);
    if (p->right) count += free_bptree (p->right);
    p->left = p->right = p->up = NULL;
    free_node (p);
  }

  return count;
}
```

```
void btree_bounding (sp)
  search_ *sp;
{ node_ *p, *q;
  long count = 0;

  q = p = sp->open.btree;
  while (p) {
    if (is_bounded(p, sp)) {
      count++;
      if (p == sp->open.btree) {
        sp->open.btree = p->next;
        free_node(p);
        q = p = sp->open.btree;
      }
      else {
        q->next = p->next;
        free_node(p);
        p = q->next;
      }
    }
    else {
      p = (q = p)->next;
    }
  }
  sp->stat.active -= count;
}


void list_bounding (sp)
  search_ *sp;
{ node_ *p, *q;
  long count = 0;

  q = p = sp->open.list;
  while (p) {
    if (is_bounded(p, sp)) {
      count++;
      if (p == sp->open.list) {
        sp->open.list = p->next;
        free_node(p);
        q = p = sp->open.list;
      }
      else {
        q->next = p->next;
        free_node(p);
        p = q->next;
      }
    }
    else {
      p = (q = p)->next;
    }
  }

  sp->stat.active -= count;
}


void young_bounding (sp, depth)
  search_ *sp;
  int depth;
{ node_ *p, *q;
  long count = 0;

  q = p = (sp->band.tree+depth)->young;
  while (p) {
    if (is_bounded(p, sp)) {
      count++;
      if (p == (sp->band.tree+depth)->young) {
        (sp->band.tree+depth)->young = p->next;
        free_node(p);
        q = p = (sp->band.tree+depth)->young;
      }
      else {
        q->next = p->next;
        free_node(p);
        p = q->next;
      }
    }
    else {
      p = (q = p)->next;
    }
  }

  sp->stat.active -= count;
}


void set_search_message (msgp, style, rp, alg_entry, rt_pf, st_pf)
  search_message_ *msgp;
  search_style_ style;
  regress_ *rp;
  yesno_ alg_entry, rt_pf, st_pf;
{
  msgp->style = style;
  msgp->rp = rp;
  msgp->alg_entry = alg_entry;
  msgp->rt_pf = rt_pf;
  msgp->st_pf = st_pf;
  msgp->feasible = NO;
}


is_feasible_or_equiv (node)
  node_ *node;
{
  return is_feasible (node);
}

::::::::::::::
kernel/stat.c
::::::::::::::

void update_stat (sp)
  search_ *sp;
{ stat_ *p = &(sp->stat);
  long interval;

  /* virtual time, space are set in 'insert'; virtual cst is set here */
  p->v_cst += tofloat(p->active);

  /* real time, space, cst are set here */
  times(&break_time);
  p->time += (interval = (break_time.tms_utime - last_time.tms_utime));
  last_time = break_time;
  p->r_cst += ((float) (p->active * node_conf.node_size * interval));
  /* update max_active, if necessary */
  p->max_active = max(p->max_active,1+p->active);
} /* udpate_stat */


long get_time (sp)
  search_ *sp;
{
#ifdef TIME_IS_GEN
  return ((cmd.count == REAL) ? sp->stat.time : sp->stat.generated);
#else
  return ((cmd.count == REAL) ? sp->stat.time : sp->stat.expanded);
#endif
}


long get_space (sp)
  search_ *sp;
{ long space = sp->stat.active;

  if (cmd.count == REAL) space *= node_conf.node_size;
  return space;
}


long get_max_space (sp)
  search_ *sp;
{ long max_space = sp->stat.max_active;

  if (cmd.count == REAL) max_space *= node_conf.node_size;
  return max_space;
}


float get_cst (sp)
  search_ *sp;
{ return ((cmd.count == REAL) ? sp->stat.r_cst : sp->stat.v_cst); }


long get_pruned (sp)
  search_ *sp;
{ stat_ *p = &(sp->stat);

  return (p->bounded + p->dominated + p->bounding + p->dominating);
}


void merge_stat_to_parent (childp, parentp)
  search_ *childp, *parentp;
{ stat_ *p = &(parentp->stat), *c = &(childp->stat);

  p->generated += c->generated;
  p->expanded += c->expanded;
  p->feasible += c->feasible;
  p->infeasible += c->infeasible;
  p->active = c->active;        /* active space is not additive */
  p->bounded += c->bounded;
  p->dominated += c->dominated;
  p->bounding += c->bounding;
  p->dominating += c->dominating;
  p->hard_bounded += c->hard_bounded;
  if (c->max_active > p->max_active) p->max_active = c->max_active;
  p->time += c->time;
  p->v_cst += c->v_cst;
```

```
  p->r_cst += c->r_cst;
  p->unix_utime += c->unix_utime;
  p->unix_stime += c->unix_stime;
}


void merge_approx_to_parent (childp, parentp)
  search_ *childp, *parentp;
{ put_achieved_approx(parentp, (parentp->approx.run_time = get_approx(childp))); }


void merge_sol_to_parent (childp, parentp)
  search_ *childp, *parentp;
{
  if (parentp->glub > childp->glub) {
    parentp->glub = childp->glub;
    parentp->incumbent = childp->incumbent;
  }
  if (parentp->gllb < childp->gllb) parentp->gllb = childp->gllb;
}


void inherit_sol_from_parent (childp, parentp)
  search_ *childp, *parentp;
{
  childp->incumbent = parentp->incumbent;
  childp->glub = childp->x_root_ub = parentp->glub;
  childp->gllb = childp->x_root_lb = parentp->gllb;
}


float get_offset_rt_approx (sp, offset_sp)
  search_ *sp, *offset_sp;
{ float rt_approx;

  rt_approx = get_approx(sp);
  if (offset_sp) rt_approx = (rt_approx < offset_sp->approx.achieved) ?
                             rt_approx : offset_sp->approx.achieved;
  return rt_approx;
}


long get_offset_time (sp, offset_sp)
  search_ *sp, *offset_sp;
{
  if (offset_sp) return (get_time (sp) + get_time (offset_sp));
  else return get_time (sp);
}


long get_offset_space (sp, offset_sp)
  search_ *sp, *offset_sp;
{ long space;

  if (offset_sp) space = sp->stat.active + offset_sp->stat.active;
  else           space = sp->stat.active;
  if (cmd.count == REAL) space *= node_conf.node_size;
  return space;
}


float get_offset_cst (sp, offset_sp)
  search_ *sp, *offset_sp;
{
  if (cmd.count == REAL)
       if (offset_sp) return (sp->stat.r_cst + offset_sp->stat.r_cst);
       else return sp->stat.r_cst;
  else if (offset_sp) return (sp->stat.v_cst + offset_sp->stat.v_cst);
       else return sp->stat.v_cst;
}


long get_offset_max_space (sp, offset_sp)
  search_ *sp, *offset_sp;
{ long space;

  if (offset_sp) space = (sp->stat.max_active >= offset_sp->stat.max_active) ?
                         sp->stat.max_active : offset_sp->stat.max_active;
  else           space = sp->stat.max_active;
  if (cmd.count == REAL) space *= node_conf.node_size;
  return space;
}


long get_offset_gen (sp, offset_sp)
  search_ *sp, *offset_sp;
{
  return (((offset_sp) ? (sp->stat.generated + offset_sp->stat.generated) :
                         (sp->stat.generated)));
}
```

```
Fri Jan 31 09:59:58 CST 1992
::::::::::::::
open/bandlist.c
::::::::::::::

#define young_list(sp,depth)     (sp->band.tree+depth)->young
#define adult_count(sp,depth)    (sp->band.tree+depth)->count
#define bandwidth(sp,depth)      (sp->band.tree+depth)->width


extern node_ *btree_delete ();
void btree_insert ();

node_ *young_delete ();
void young_insert ();
#ifdef SWAP_BAND_NODE
node_ *swap_band_node ();
#endif


node_ *get_band_node (sp)
    search_ *sp;
{   node_ *node;
    tree_ *treep;
    int i, depth;
    long sum = 0;

#ifdef PEEK_ACTIVE_LIST
printf ("NEW STATISTICS\n");
for (i = 0, node = sp->open.btree; node; node = node->next) i++;
printf ("sp->open.btree=%d\n", i);
sum += i;
for (depth = sp->band.ceiling_depth; depth <= sp->band.floor_depth; depth++) {
    for (i = 0, node = young_list (sp, depth); node; node = node->next) i++;
    printf ("depth=%d, young=%d\n", depth, i);
    sum += i;
}
printf ("sp->stat.active=%d, sum=%d\n", sp->stat.active, sum);
#endif

    if (sp->stat.active <= 0) sp->stat.active = 0;
    else sp->stat.active --;

    if ((node = btree_delete (sp))) return node;

    /** update ceiling **/
    for (depth = sp->band.ceiling_depth; depth <= sp->band.floor_depth; depth++)
        if (young_list (sp, depth)) break;
    sp->band.ceiling_depth = depth;

    while (1) {
        /** try this depth, include young nodes into band **/
        treep = sp->band.tree + sp->band.floor_depth;
        treep->count = 0;
        for (i = 0; i < treep->width; i++)
            if ((node = young_delete (sp, sp->band.floor_depth))) {
                btree_insert (node, sp);
                treep->count ++;
            }
            else break;

        /** check if new adult nodes exist **/
        if ((node = btree_delete (sp))) return node;

        /** unfortunately, there is no new adult node **/
        /** check young nodes less deep in the search tree **/
        sp->band.floor_depth--;
        if (sp->band.floor_depth < sp->band.ceiling_depth) {
            sp->stat.active = 0;
            return NULL;
        }
    }
}


void put_band_node (children, sp)
    node_ *children;
    search_ *sp;
{   node_ *child, *node, *new_node;
    tree_ *treep;
    int count, depth, num, i;

    if (sp->stat.active < 0) sp->stat.active = 0;
    if (children == NULL) return;

    depth = children->depth;
    if (depth > sp->band.floor_depth) {
        sp->band.floor_depth = depth;
        if (depth > MaxDepth) {
            num = 2 * MaxDepth + 1;
            treep = (tree_ *) malloc (num * sizeof (tree_));
            for (i = 0; i <= MaxDepth; i++) {
                (treep+i)->young = (sp->band.tree+i)->young;
                (treep+i)->count = (sp->band.tree+i)->count;
                (treep+i)->width = (sp->band.tree+i)->width;
            }
            for (i = MaxDepth+1; i < num; i++)
                init_tree_struct (treep, i, sp->band.init_width);

            free (sp->band.tree);
            sp->band.tree = treep;
            MaxDepth = (-- num);
        }
    }

    while ((child = children)) {
        sp->stat.active ++;
        children = children->brother;
        young_insert (child, depth, sp);
    }

    count = adult_count (sp, depth);
    while (count < bandwidth (sp, depth))
        if ((node = young_delete (sp, depth))) {
            btree_insert (node, sp);
            count++;
        } else break;
    adult_count (sp, depth) = count;

#ifdef SWAP_BAND_NODE
    while ((node = young_delete (sp, depth))) {
        new_node = swap_band_node (node, sp);
        young_insert (new_node, depth, sp);
        if (node == new_node) break;
    }
#endif
}
```

```
#define young_sort_key(node)    (node->lowb)

/** insert a node into a b+ tree, ascendingly **/
void young_insert (node, depth, sp)
    node_ *node;
    int depth;
    search_ *sp;
{   node_ *p, *q;
    domain node_key;

    if (young_list (sp, depth)) {
        node_key = young_sort_key (node);
        for (q = p = young_list (sp, depth); p; p = (q = p)->next) {
            if (node_key < young_sort_key(p)) {
                if (p == young_list (sp, depth)) {
                    (young_list (sp, depth) = node)->next = p;
                    return;
                } else {
                   (q->next = node)->next = p;
                   return;
                } /* if-then-else */
            } /* if-then */
        } /* for */
        if (! p) (q->next = node)->next = NULL;
    }
    else (young_list (sp, depth) = node)->next = NULL;
}


/** delete a node from a b+ tree **/
node_ *young_delete (sp, depth)
    search_ *sp;
    int depth;
{   node_ *q;

    if (young_list (sp, depth))
        young_list (sp, depth) = (q = young_list (sp, depth))->next;
    else return NULL;

    return q;
}


int is_young_empty (sp, depth)
    search_ *sp;
    int depth;
{   return (young_list (sp, depth) == NULL); }

#undef young_list
#undef adult_count
#undef bandwidth
#undef young_sort_key

::::::::::::::
open/bptree.c
::::::::::::::

#ifdef BPLUS_TREE

#define bptree_sort_key(node)    (node->lowb)

/* insert a node into a b_plus tree, ascendingly */
void bptree_insert (node, sp)
  node_ *node;
  search_ *sp;
{ node_ *p;
  domain node_key;

  node->left = node->right = NULL;

  if (sp->strategy != BAND)
    if (sp->stat.active < 0) sp->stat.active = 0;
    else sp->stat.active++;

  if (sp->open.bptree) {
    node_key = bptree_sort_key(node);

    for (p = sp->open.bptree; p; ) {
      if (node_key < bptree_sort_key(p)) {
        if (p->left) p = p->left;
        else {
          p->left = node;
          node->up = p;
          return;
        }
      } else {
        if (p->right) p = p->right;
        else {
          p->right = node;
          node->up = p;
          return;
        }
      }
    }

    error ("bptree_insert: Internal error");
  }

  sp->open.bptree = node;
  node->up = NULL;
}



/* delete a node from a b_plus tree */
node_ *bptree_delete (sp)
  search_ *sp;
{ node_ *p;

  if (sp->open.bptree) {
    for (p = sp->open.bptree; p->left; p = p->left) ;
    if (p == sp->open.bptree) {
      sp->open.bptree = p->right;
      if (p->right) p->right->up = NULL;
    } else {
      p->up->left = p->right;
      if (p->right) p->right->up = p->up;
    }
    p->left = p->right = p->up = NULL;
    return p;
  }

  return NULL;
}
```

```
int is_bptree_empty (sp)
  search_ *sp;
{ return (sp->open.bptree == NULL); }


#undef  bptree_sort_key(node)

#else BPLUS_TREE

void bptree_insert (node, sp)
  node_ *node;
  search_ *sp;
{ btree_insert (node, sp); }

node_ *bptree_delete (sp)
  search_ *sp;
{ return btree_delete (sp); }

int is_bptree_empty (sp)
  search_ *sp;
{ return is_btree_empty (sp); }

#endif

::::::::::::::
open/btree.c
::::::::::::::


#define btree_sort_key(node)     (node->lowb)

/* insert a node into a virtual b+ tree, ascendingly */
void btree_insert (node, sp)
  node_ *node;
  search_ *sp;
{ node_ *p, *q;
  domain node_key;

  if (sp->strategy != BAND)
    if (sp->stat.active < 0) sp->stat.active = 0;
    else sp->stat.active++;

  if (sp->open.btree) {
    node_key = btree_sort_key(node);
    for (q = p = sp->open.btree; p; p = (q = p)->next) {
      if (node_key < btree_sort_key(p)) {
        if (p == sp->open.btree) {
          (sp->open.btree = node)->next = p;
          return;
        }
        else {
          (q->next = node)->next = p;
          return;
        } /* if-then-else */
      } /* if-then */
    } /* for */
    if (! p) (q->next = node)->next = NULL;
  }
  else (sp->open.btree = node)->next = NULL;
}


/* delete a node from a b+ tree */
node_ *btree_delete (sp)
  search_ *sp;
{ node_ *q;

  if (sp->open.btree) sp->open.btree = (q = sp->open.btree)->next;
  else {
    if (sp->strategy != BAND) sp->stat.active = 0;
    return NULL;
  }

  if (sp->strategy != BAND)
    if(sp->stat.active <= 0) sp->stat.active = 0;
    else sp->stat.active--;

  return q;
}


int is_btree_empty (sp)
  search_ *sp;
{ return (sp->open.btree == NULL); }


#ifdef SWAP_BAND_NODE
node_ *swap_band_node (node, sp)
  node_ *node;
  search_ *sp;
{ node_ *p, *q, *temp;
  domain node_key;

  if (sp->open.btree) {
    node_key = btree_sort_key(node);
    for (q = p = sp->open.btree; p; p = (q = p)->next) {
      if (node->depth == p->depth && node_key < btree_sort_key(p)) {
        if (p == sp->open.btree) (sp->open.btree = node)->next = p->next;
        else (q->next = node)->next = p->next;
        temp = p; p = node; node = temp;  /** swap **/
        node_key = btree_sort_key(node);
      } /* if-then */
    } /* for */
  }

  return node;
}

#endif

#undef  btree_sort_key(node)

::::::::::::::
open/list.c
::::::::::::::


/* insert a node into an ordered list , ascendingly */
void list_insert (node, sp)
  node_ *node;
  search_ *sp;
{ node_ *p, *q;
  domain node_key;

  if (sp->stat.active < 0) sp->stat.active = 0;
  else sp->stat.active++;

  if (sp->open.list) {
    node_key = sort_key(node, sp);
    for (q = p = sp->open.list; p; p = (q = p)->next) {
      if (node_key < sort_key(p, sp)) {
        /** insert **/
        if (p == sp->open.list) (sp->open.list = node)->next = p;
        else (q->next = node)->next = p;
        return;
      }
    }
    if (! p) (q->next = node)->next = NULL;
  }
  else (sp->open.list = node)->next = NULL;
}


/* delete a node from a linked list */
node_ *list_delete (sp)
  search_ *sp;
{ node_ *q = NULL;

  if (sp->open.list) sp->open.list = (q = sp->open.list)->next;
  else { sp->stat.active = 0; return NULL; }
  if (sp->stat.active < 0) sp->stat.active = 0;
  else sp->stat.active--;
  return q;
}


int is_list_empty (sp)
  search_ *sp;
{ return (sp->open.list == NULL); }

::::::::::::::
open/pool.c
::::::::::::::

node_ *get_search_node_from_pool ()
{ int offset, *pool, i;
  node_ *p;

  /* if the pool of search nodes are empty, then
   * allocate AllocationSize search nodes at a time to avoid
   * spreadout of search nodes to entire virtual
   * memory.
   */

#ifdef DEBUG
if (cmd.debug >= 4) printf("enter get_search_node_from_pool\n");
#endif

  offset = node_conf.node_size / sizeof(int);
  if (! pool_manager) {  /* request space from OS */

#ifdef DEBUG
if (cmd.debug >= 4) printf("allocate main body\n");
#endif
    /* allocate entire chunk of search nodes, including
     * main body and associative regions
     */
    pool = (int *) malloc(AllocationSize * node_conf.node_size);
    pool_manager = (node_ *) pool;

    for (i = 0; i < AllocationSize; i++) {
      p = ((node_ *) (pool + offset * i));
      p->next = ((node_ *) (pool + offset * (i + 1)));
    }
    p->next = NULL;

    pd_region();
  }

#ifdef DEBUG
if (cmd.debug >= 4) printf("now, get a search node\n");
#endif

  /* now, pool has free search nodes */
  pool_manager = (p = pool_manager)->next;
  NumNodesTaken++;
  return p;
}


void dispose_search_node_to_pool (node)
  node_ *node;
{
  node->next = pool_manager;
  pool_manager = node;
  NumNodesTaken--;
}

::::::::::::::
open/sort.c
::::::::::::::

void insert (node, sp)
  node_ *node;
  search_ *sp;
{
  switch (sp->strategy) {
    case GBB:  list_insert(node, sp);  break;
    case BFS:  bptree_insert(node, sp); break;
    case DFS:  stack_push(node, sp);   break;
    case GDFS: list_insert(node, sp);  break;
    case BAND: node->brother = NULL;
               put_band_node (node, sp); break;
    default:   list_insert(node, sp);  break;
  }
}


node_ *delete (sp)
  search_ *sp;
{
  switch (sp->strategy) {
    case GBB:  return list_delete(sp);    break;
    case BFS:  return bptree_delete(sp);   break;
    case DFS:  return stack_top(sp);      break;
    case GDFS: return list_delete(sp);    break;
    case BAND: return get_band_node (sp); break;
    default:   return list_delete(sp);    break;
  }
}


domain sort_key (node, sp)
  node_ *node;
  search_ *sp;
{ domain key;
```

```
  if (sp->strategy == BAND) return node->lowb;

  if (problem.domain == INT)
    key = (sp->strategy == BFS) ? (node->lowb) : (HUGE_DEPTH - node->depth);
  else
    key = (sp->strategy == BFS) ? (node->lowb) : ((float) (HUGE_DEPTH - node->depth));
  return key;
}


int is_open_empty (sp)
  search_ *sp;
{ return (sp->stat.active == 0); }

::::::::::::::
open/stack.c
::::::::::::::


void stack_push (node, sp)
  node_ *node;
  search_ *sp;
{
  if (sp->stat.active < 0) sp->stat.active = 0;
  else sp->stat.active++;
  node->next = sp->open.stack;
  sp->open.stack = node;
}


int is_stack_empty (sp)
  search_ *sp;
{ return (sp->open.stack == NULL); }


node_ *stack_top (sp) search_ *sp; { return sp->open.stack; }

void stack_pop (sp)
  search_ *sp;
{ node_ *temp;

  if (temp = sp->open.stack) {
    if (sp->stat.active < 0) sp->stat.active = 0;
    else sp->stat.active--;
    sp->open.stack = temp->next;
    free_node(temp);
  }
  else sp->stat.active = 0; /* empty stack */
}


node_ *stack_bottom (sp)
  search_ *sp;
{ node_ *p;

  if ((p = sp->open.stack) == NULL) return NULL;
  while (p->next) p = p->next;
  return p;
}
```

```
Thu Jan 30 15:45:42 CST 1992
::::::::::::::
solver/ats.pgm/define.h
::::::::::::::
#ifndef __ATS_define_h_
#define __ATS_define_h_

#ifndef SYM_TSP
#ifndef ASYM_TSP
#define ASYM_TSP
#endif
#endif

typedef float   domain;
#define Domain  FLOAT

#define AllocationSize          256

#define PDSI_PART       \
        int     *open;    /* open cities, same as *live if this node is new, \
                             and it's different from *live if compound node. \
                             (not considered yet) */ \
        int     nvisit;   /* num of cities visited already */ \
        int     *visited; /* visited cities */


/* for problem generation */
#define MaxProblemSize  200
#define MaxLinkSize     (MaxProblemSize * MaxProblemSize)

/* geographical limit of graph */
#define GeoLimit                100

typedef struct
{
        int     city[MaxProblemSize];
} solution_;

#define encodeW(x,y)    (x * nCities + y)
#define decodeW(road)   *(WEIGHT + road)
#define Weight(x,y)     *(WEIGHT + encodeW(x,y))
#define sortW(index)    *(SORTW + index)
#define decodeX(road)   ((int) (road / nCities))
#define decodeY(road)   (road % nCities)

#endif  __ATS_define_h_

::::::::::::::
solver/ats.pgm/abc.c
::::::::::::::

#ifdef SYM_TSP
float   complFactor = 1.0;      /* completeness factor of out degree */
#else
float   complFactor = 0.05;     /* completeness factor of out degree */
#endif

int     nCities;                /* # of cities */
int     nRoads;                 /* # of roads */
float   *WEIGHT = NULL;         /* weights of edges*/
int     *SORTW = NULL;          /* sorted version */
int     ubArray[MaxProblemSize];

int     OpenCity[MaxProblemSize];       /* temp storage for open */
int     LiveCity[MaxProblemSize];       /* temp storage for live */
int     VisitedRoad[MaxProblemSize];    /* temp storage for tracing roads */

::::::::::::::
solver/ats.pgm/bound.c
::::::::::::::
domain ats_eval_lowb (), ats_eval_upb ();


/**  Lower bound is calculated by spanning-tree heuristic **/

void evaluate_lower_bound (node)
  node_ *node;
{
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval lowb\n");
#endif

  node->lowb = (is_pre_goal (node)) ? node->g_cost : ats_eval_lowb (node);
}


domain ats_eval_lowb (node)
  node_ *node;
{ domain cost = node->g_cost;
  int num_live_cities = nCities - node->nvisit;
  int num_roads = node->nvisit - 1;      /* visited roads */
  int entity = node->entity;
  int i, j, road, x, y, s1, s2, the_first_not_done, the_last_not_done;
  int vertex[MaxProblemSize];

  /* find which cities not visited yet */
  trace_live(node);
  /** LiveCity[i] = 1 ==> live; otherwise, 0. **/
  LiveCity[entity] = 1;
  num_live_cities++;
  for (i = 0; i < nCities; i++) vertex[i] = LiveCity[i] ? i : -1;

  /* find which roads have been traversed and put into VisitedRoad */
  trace_road(node);

#ifdef DEBUG
if (cmd.debug >= 5) {
  printf("live cities=%d\n", num_live_cities);
  for (i = 0; i < nCities; i++) printf("%d-th city=%d\n", i, LiveCity[i]);
  printf("num_roads=%d\n", num_roads);
  for (i = 0; i < num_roads; i++) printf("%d-th road=%d\n", i, VisitedRoad[i]);
}
#endif

  /* spanning-tree cost */
  for (i = 0; (i < nRoads) && (num_live_cities > 0); i++) {
    road = sortW(i);
    if (! is_member(road, VisitedRoad, 0, num_roads)) {
      x = decodeX(road);
      y = decodeY(road);
      s1 = vertex[x];
      s2 = vertex[y];
      if ((s1 != -1) && (s2 != -1) && (s1 != s2)) {
        num_live_cities--;
        cost += decodeW(road);
        for (j = 0; j < nCities; j++)
          if (vertex[j] == s2) vertex[j] = s1;
    }
    }
  }

  /** cost of connecting spanning tree and the tree explored already **/
  for (i = 0; i < nRoads; i++) {
    road = sortW (i);
    if (! is_member (road, VisitedRoad, 0, num_roads)) {
      x = decodeX (road);
      y = decodeY (road);
      if (x == 0)
        if (LiveCity[y]) { cost += decodeW (road); break; }
        else if (y == 0)
          if (LiveCity[x]) { cost += decodeW (road); break; }
    }
  }

  return cost;
}


/**  Upper bound is calculated by the hill-climbing method if applicable,
 ** otherwise, a greedy back-tracking method is applied. **/

solution_ *evaluate_upper_bound (node, sol)
  node_ *node;
  solution_ *sol;
{ int i, num_cities = node->nvisit;

#ifdef DEBUG
if (cmd.debug >= 4) printf("eval upb\n");
#endif

  if (is_feasible(node)) {
    if (num_cities != nCities) error("evaluate_upper_bound");
    for (i = 0; i < num_cities; i++) sol->city[i] = *(node->visited + i);
    node->upb = node->g_cost;
  }
  else node->upb = ats_eval_upb(node, sol);
  return sol;
}


domain ats_eval_upb (node, sol)
  node_ *node;
  solution_ *sol;
{ int x, y, who, i, num_cities = node->nvisit;
  domain cost = node->g_cost, temp, temp0, best, bk_find_sol();

  for (i = 0; i < num_cities; i++) sol->city[i] = *(node->visited + i);
#ifdef DEBUG
if (cmd.debug >= 5)
  for (i = 0; i < num_cities; i++) printf("i=%d, city=%d\n", i, sol->city[i]);
#endif

  trace_live(node);

  x = node->entity;
  while (1) {
    /* find the best next city */
    best = HUGE_FLOAT;
    who = -1;
    for (y = 1; y < nCities; y++) {
      if (LiveCity[y] && (temp = Weight(x,y)) != problem.huge) {
        if (num_cities == nCities - 1)
          if ((temp0 = Weight(y,0)) != problem.huge) temp += temp0;
          else continue;
        if (best > temp) { best = temp; who = y; }
      }
    }
#ifdef DEBUG
if (cmd.debug >= 5) printf("num_cities=%d, who=%d\n", num_cities, who);
#endif
    /* record the best next city */
    if (who == -1) {
      /** OLD VERSION, NOT APPROPRIATE.
       ** if (node == ROOT) return bk_find_sol(node, sol);
       ** else return problem.huge; **/
      return problem.huge;
    }
    sol->city[num_cities++] = who;
    if (num_cities == nCities) return (cost + best);
    LiveCity[who] = 0;
    cost += best;
    x = who;
  }
}


domain bk_find_sol (node, sol)
  node_ *node;
  solution_ *sol;
{ int num_cities = node->nvisit;
  int x = node->entity;
  int next = 1, y, i;
  domain cost = node->g_cost;
  domain temp, temp0;

#ifdef DEBUG
if (cmd.debug >= 4) printf("enter bk_find_sol\n");
if (cmd.debug >= 5)
  for (i = 0; i < node->nvisit; i++)
    printf("i=%d, city=%d\n", i, sol->city[i]);
#endif

  trace_live(node);

  while (num_cities < nCities) {
    for (y = next; y < nCities; y++) {
      if (LiveCity[y] && (temp = Weight(x,y)) != problem.huge) {
        if (num_cities == nCities - 1)
          if ((temp0 = Weight(y,0)) != problem.huge) temp += temp0;
          else continue;
#ifdef DEBUG
if (cmd.debug >= 5)
  printf("num_cities=%d, city=%d\n", num_cities, y);
#endif
        cost += temp;
        LiveCity[y] = 0;
        sol->city[num_cities++] = (x = y);
        next = 1;
        break;
      }
    }
    if (y >= nCities) {
      if (node->nvisit == num_cities) return problem.huge;  /* infeasible */
      /* recover to prev stage */
      y = sol->city[--num_cities];
      LiveCity[y] = 1;
      x = sol->city[num_cities - 1];
      cost -= Weight(x,y);
      next = y + 1;
    }
  }
  return cost;
}


trace_live (node)
  node_ *node;
{ int i;

  /* reset LiveCity array */
  for (i = 0; i < nCities; i++) LiveCity[i] = 1;

  /* mark real live city */
  for (i = 0; i < node->nvisit; i++) LiveCity[*(node->visited + i)] = 0;
}


trace_road (node)
  node_ *node;
{ int i, j, city_1, city_2;

  if (node->nvisit >= 2) {
    for (i = 0, j = 1; j < node->nvisit; i++, j++) {
      city_1 = *(node->visited + i);
      city_2 = *(node->visited + j);
      VisitedRoad[i] = encodeW(city_1, city_2);
    }
  }
}

::::::::::::::
solver/ats.pgm/gen.c
::::::::::::::

/* ******************************************************************* *
 * Traveling Salesperson Problem Generator:                            *
 * 1. incomplete graph.                                                *
 * 2. asymmetric graph.                                                *
 * 3. triangular inequality is satisfied for existing arcs.            *
 *    (existing arcs == arcs of finite distance)                       *
 * 4. completeness factor. [0,1]                                       *
 * ******************************************************************* */


void gen_sample_problem ()
{ float  best, tmp, dx, dy, dist, x[MaxProblemSize], y[MaxProblemSize];
  int i, j, best_one, the_end, the_one;

#ifndef SYM_TSP
  FILE *fp, *fopen ();
  No_Upper_Bound = YES;
#endif

  nCities = problem.size;

  /** build the map **/
  for (i = 0; i < nCities; i++) {
    /* create a new city */
    x[i] = gen_random_int (0, GeoLimit);
    y[i] = gen_random_int (0, GeoLimit);

    /* check if the new city overlaps other cities */
    for (j = 0; j < i; j++)
      if (x[i] == x[j])
        while (y[i] == y[j])
          /* repeat trying until no overlap */
          y[i] = gen_random_int (0, GeoLimit);
  }

  /** init weight matrix **/
  for (i = 0; i < nCities; i++)
    for (j = 0; j < nCities; j++)
      Weight(i,j) = problem.huge;

  /** calc distance **/
  nRoads = 0;
  for (i = 0; i < nCities; i++) {
    for (j = i+1; j < nCities; j++) {
      if (gen_random_float () <= complFactor ||
          j == i+1 ||
          (i == 0 && j == nCities-1)) {
        dx = x[i] - x[j];
        dy = y[i] - y[j];
        dist = dx * dx + dy * dy;
        Weight(i,j) = Weight(j,i) = (domain) sqrt (todouble (dist));
        sortW(nRoads) = encodeW(i,j);
        nRoads++;
      }
    }
  }

  /* establish the sorted version of weights of edges */
  for (i = 0; i < nRoads-1; i++) {
    best_one = i;
    best = decodeW(sortW(best_one));
    for (j = i+1; j < nRoads; j++)
      if (decodeW(sortW(j)) < best) { best_one = j; best = decodeW(sortW(j)); }
    tmp = sortW(i); sortW(i) = sortW(best_one); sortW(best_one) = tmp;
  }

#ifdef DEBUG
if (cmd.debug >= 10) print_conf(stdout);
if (cmd.debug >= 10) {
  printf("\n");
  for (i = 0; i < nRoads; i++)
    printf("(sortW[%d] = %d, %f), ", i, sortW(i), decodeW(sortW(i)));
}
#endif

  /* PDSD size */
  set_node_size((nCities + nCities) * sizeof(int));

#ifndef SYM_TSP
  fp = fopen (".ats", "a");
  fprintf (fp, "ATSP(%d,%d,%d)\n", problem.size, nRoads, problem.seed);
  fclose (fp);
#endif
}

::::::::::::::
solver/ats.pgm/search.c
::::::::::::::

typedef float decomp_;
decomp_ decomp_h_fun ();
```

```
void iipd_init () {
  int size = problem.size;

  /* create weight matrix */
  WEIGHT = (float *) malloc(size * size * sizeof(float));
  SORTW = (int *) malloc(size * size * sizeof(int));
}


void idpd_init () { }

int is_dominated (node) node_ *node; { return 0; }


/* node allocation routine */
node_ *allocate_node (parentp, entity, g_cost)
  node_ *parentp;
  int entity;
  domain g_cost;
{ node_ *p;
  int i;

  p = get_search_node_from_pool();
  init_node_struct(p, parentp);

  p->entity = entity;
  p->g_cost = g_cost;

  /* clean thses value fields */
  if (parentp) {
    p->lowb = parentp->lowb;
    p->upb = parentp->upb;
  } else {
    p->lowb = - problem.huge;
    p->upb = problem.huge;
  }

  /* init *visited associate */
  if (parentp) {
    for (i = 0; i < parentp->nvisit; i++)
      *(p->visited + i) = *(parentp->visited + i);
    p->nvisit = parentp->nvisit + 1;
    *(p->visited + parentp->nvisit) = entity;
  }
  else {
    p->nvisit = 1;
    *(p->visited) = entity;
  }

  /* init *open associate */
  trace_live(p);
  for (i = 0; i < nCities; i++)
    *(p->open+i) = (LiveCity[i] && Weight(entity,i) != problem.huge) ? 1 : 0;

  return p;
}


/* allocate associates for newly allocated search nodes */
void pd_region ()
{ int offset, *pool, visited_offset, open_offset, *a, i;
  node_ *p;

  pool = (int *) pool_manager;
  offset = node_conf.node_size / sizeof(int);
  visited_offset = sizeof(node_) / sizeof(int);
  open_offset = visited_offset + nCities;

  for (i = 0; i < AllocationSize; i++) {
    p = ((node_ *) (a = pool + offset * i));
    p->visited = a + visited_offset;    /* allocate *visited frame */
    p->open = a + open_offset;          /* allocate *open frame */
  }
}


void reset_sol_buf (s)
  solution_ *s;
{ int i;

  for (i = 0; i < problem.size; i++) s->city[i] = 0;
}


/* for debug */
void print_conf (fp)
  FILE *fp;
{ int i, j;

  fprintf(fp, "nCities=%d,  nRoads=%d\n", nCities, nRoads);
  for (i = 0; i < nCities; i++) {
    fprintf(fp, "\n");
    for (j = 0; j < nCities; j++)
      if (Weight(i,j) != problem.huge)  fprintf(fp, " %6f ", Weight(i,j));
      else                              fprintf(fp, " huge ");
  }
  fprintf(fp, "\n");
}


void evaluate_solution (sp)
  search_ *sp;
{ solution_ *sol;
  int i;

  if (sp) sol = sp->incumbent;
  else error("evaulate_solution: internal error: sp is nil");
  printf("\nSOLUTION:\n");
  for (i = 0; i < nCities; i++) printf("%d-th city is  %d\n", i, sol->city[i]);
}
```

```
decomp_ dh_val[MaxProblemSize];
node_ *ch_arr[MaxProblemSize];


node_ *expand (node, child_type, nchild)
  node_ *node;
  child_ child_type;
  int nchild;
{ int entity = node->entity;
  domain g_cost = node->g_cost;
  node_ *new_list = NULL;
  node_ *child, *next_child;
  int i, count = 0, who;
  domain w;
  decomp_ best;

  switch (child_type) {
  case ALL_CHILDREN:  nchild = HUGE_INT;
  case NEXT_N_CHILDREN:

    /** init **/
    for (i = 0; i < nCities; i++) {
      dh_val[i] = HUGE_FLOAT;
      ch_arr[i] = NULL;
    }

    /** sprout all necessary children **/
    for (i = 1; i < nCities; i++)
      if (*(node->open+i)) {
        *(node->open+i) = 0;
        w = Weight (entity, i);
        if (w != problem.huge) {
          ch_arr[i] = allocate_node (node, i, g_cost+w);
          dh_val[i] = decomp_h_fun (node, ch_arr[i]);
          if (++count >= nchild) break;
        }
    }

    /** sort all these children by decomp heuristic values **/
    while (count-- > 0) {
      best = HUGE_FLOAT;
      who = -1;
      for (i = 1; i < nCities; i++)
        if (best > dh_val[i]) { best = dh_val[i]; who = i; }
      ch_arr[who]->next = new_list;
      new_list = ch_arr[who];
      dh_val[who] = HUGE_FLOAT;
    }

    /** reverse new_list **/
    next_child = new_list;
    new_list = NULL;
    while ((child = next_child)) {
      next_child = child->next;
      child->next = NULL;
      child->brother = new_list;
      new_list = child;
    }

    return new_list;
    break;

  case NEXT_CHILD:
    for (i = 1; i < nCities; i++)
      if (*(node->open+i)) {
        *(node->open+i) = 0;
        w = Weight (entity, i);
        if (w != problem.huge) return allocate_node (node, i, g_cost+w);
      }
    break;

  default: break;
  }

  return NULL;
}


decomp_ decomp_h_fun (node, child)
  node_ *node, *child;
{
  return Weight (node->entity, child->entity);
}


int is_infeasible (node)
  node_ *node;
{ int v;

  /* an infeasible node must be a compound node in the complete graph,
   * but could be a simple node in incomplete graph
   */
  if (node->nvisit == nCities) return 0;
  for (v = 0; v < nCities; v++)
    if (*(node->open+v)) return 0;
  return 1;
}


int is_feasible (node)
  node_ *node;
{
  if (node->nvisit == nCities)
    return ((Weight (node->entity, 0) == problem.huge) ? 0 : 1);
  else return 0;
}


int is_pre_goal (node)
  node_ *node;
{ int v, edge;

  if (node->nvisit == nCities)
    if (Weight (node->entity, 0) != problem.huge) {
      node->g_cost += Weight (node->entity, 0);
      return 1;
    }
  return 0;
}

::::::::::::::
solver/ats.pgm/support.c
::::::::::::::

node_ *root_generator () { return allocate_node(NULL, 0, 0.0); }


/* iteration-independent problem-dependent init */
```

```
Thu Jan 30 15:46:10 CST 1992
::::::::::::::
solver/ks.pgm/define.h
::::::::::::::
#ifndef __node_h_
#define __node_h_

typedef int     domain;
#define Domain  INT

#define AllocationSize          256

#define PDSI_PART \
        int     w;      /* acc weight */  \
        int     *item;  /* item pointer */

#define _W_Low                  1
#define _W_Up                   1000

#define _Variance               1.5
#define MaxProblemSize          550

typedef struct
{
        int     not_in_sack[MaxProblemSize + 1];
} solution_;

extern int      U;              /* # items */
extern int      B;              /* bound of weight */
extern int      BOUND;          /* lowb sum of weight of not included items */
extern int      PP;             /* overall sum of profits */
extern int      WW;             /* overall sum of weights */
extern int      W[MaxProblemSize];      /* weights */
extern int      P[MaxProblemSize];      /* profit */
extern int      sortP[MaxProblemSize];
extern float    PdivW[MaxProblemSize];

#endif __node_h_

::::::::::::::
solver/ks.pgm/bound.c
::::::::::::::

domain ks_eval_lowb (), ks_eval_upb ();


void evaluate_lower_bound (node)
  node_ *node;
{
  node->lowb = is_feasible (node) ? node->g_cost : ks_eval_lowb (node);
}


domain ks_eval_lowb (node)
  node_ *node;
{ int w_sum = node->w, i, diff, goal = 1;
  float lookahead, ratio;

  if (w_sum >= BOUND)  /* goal node already */ return node->g_cost;

  /* find min penalty & max weight */
  for (i = 1; i <= U; i++)
    if (*(node->item + sortP[i] - 1)) {
      ratio = PdivW[sortP[i]];
      goal = 0;
      break;
    }

  if (goal) return node->g_cost;

  diff = BOUND - w_sum;
  lookahead = tofloat (diff) * ratio;
  return (node->g_cost + toint (lookahead));
}


solution_ *evaluate_upper_bound(node, sol)
  node_ *node;
  solution_ *sol;
{
  node->upb = is_feasible (node) ? node->g_cost : ks_eval_upb (node, sol);
  return sol;
}

domain ks_eval_upb (node, sol)
  node_ *node;
  solution_ *sol;
{ domain p_sum = node->g_cost;
  int w_sum = node->w, i, *itemp, *ip;

  itemp = node->item;
  for(i = 1; i <= U; i++) {
    if(w_sum < BOUND) {
      ip = itemp + sortP[i] - 1;
      if(*ip) {
        w_sum += W[sortP[i]];
        p_sum += P[sortP[i]];
      }
    }
    else break;
  }

  if(w_sum < BOUND)
    /* infeasible */
    return problem.huge;

  return p_sum;
}

::::::::::::::
solver/ks.pgm/define.c
::::::::::::::

int     U;                      /* # items */
int     B;                      /* bound of weight */
int     BOUND;                  /* lowb sum of weight of not included items */
int     PP;                     /* overall sum of profits */
int     WW;                     /* overall sum of weights */
int     W[MaxProblemSize];      /* weights */
int     P[MaxProblemSize];      /* profit */
int     sortP[MaxProblemSize];
float   PdivW[MaxProblemSize];

::::::::::::::
solver/ks.pgm/dom.c
::::::::::::::

#define d_node_ struct d_node__
```

```
d_node_ {
        int     w;
        int     cost;
        node_   *who;           /* pointer to search node */
        d_node_ *next;          /* pointer to next d_node */
} *d_node_manager = NULL, *Dom[MaxProblemSize + 2];

extern void dom_init(), d_list_free(), d_node_free(), d_node_delete();
extern d_node_ *d_node_alloc();


void dom_init ()
{ int  i;

  for (i = 0; i <= problem.size + 1; i++) {
    d_list_free(Dom[i]);
    Dom[i] = NULL;
  }
}


void d_list_free (p)
  d_node_ *p;
{ d_node_ *q;

  while (p) { q = p->next; d_node_free(p); p = q; }
}


void d_node_free (p)
  d_node_ *p;
{
  p->next = d_node_manager;
  d_node_manager = p;
}


d_node_ *d_node_alloc (node)
  node_ *node;
{ d_node_ *p;
  int i;

  /* if the pool of d nodes are empty, then */
  /* allocate 1K d nodes at a time to avoid spreadout of */
  /* d nodes to a large range of virtual pages */
  if (! d_node_manager) {
    p = d_node_manager = (d_node_ *) malloc(1024 * sizeof(d_node_));
    for (i = 0; i < 1023; i++) (p + i)->next = p + i + 1;
    (p + 1023)->next = NULL;
  /*  memory_update( 1024 * sizeof(d_node_));  */
  }

  d_node_manager = (p = d_node_manager)->next;
  p->who = node;
  p->w = node->w;
  p->cost = node->g_cost;
  return p;
}


int is_dominated (node)
  node_ *node;
{ domain g_cost = node->g_cost;
  int depth = node->depth;
```

```
  int w = node->w;
  int exist_bit = 0;
  d_node_ *p, *q, *r;

  /* even if DFS, dominance is still necessary
   * if(Search_Strategy == _DFS_) { return(0); }
   */

  /* if the dom list of this depth is empty
   * then just insert it and return not-dominated
   */
  if (! Dom[depth]) {
    (Dom[depth] = d_node_alloc(node))->next = NULL;
    return 0;
  }

  /* trace down the dom list of this depth
   * let cost be the penalty
   * j is dominated by i, if
   * w(j) <= w(i)   &&   cost(j) >= cost(i)
   */
  for (p = q = Dom[depth]; p; p = (q = p)->next) {
    if (w <= p->w) {
      if (g_cost >= p->cost) {
        /* node is dominated by p */
        if (node == p->who) { exist_bit = 1; continue;}/* dominated by itself */
        /* else,
         * node is dominated by someone else
         * locate node's d_node and delete it
         */
        else { d_node_delete(node); return 1; }
      }
    }
    else {
      /* w > p->w */
      if (! exist_bit) {
        /* allocate an entry to this dom list */
        r = d_node_alloc(node);
        if (q == Dom[depth] && q == p)  (Dom[depth] = r)->next = p;
        else (q->next = r)->next = p;
      }
      return 0;
    }
  }

  /* pass through all the elements of the dom list */
  /* all elements' w's are smaller than node's w */
  /* then the loop will end with q is the last element of the list */
  if (! exist_bit) (q->next = d_node_alloc(node))->next = NULL;

  return 0;
}


void d_node_delete (node)
  node_ *node;
{ d_node_ *p, *q;

  for (q = p = Dom[node->depth]; p; p = (q = p)->next)
    if (node == p->who) {
      if (q == Dom[node->depth] && q == p) Dom[node->depth] = p->next;
      else q->next = p->next;
      d_node_free(p);
      break;
    }
}

::::::::::::::
solver/ks.pgm/gen.c
::::::::::::::

void gen_sample_problem ()
{ int i, j, best_one, tmp, min_w, size;
  float best, b_float;

  /* generate the problem size randomly */
  U = size = problem.size;

  /* generate the problem parameters randomly */
  min_w = _W_Up + 1;
  WW = PP = W[0] = P[0] = 0;
  for (j = 1; j <= U; j++) {
    WW += (W[j] = gen_random_int(_W_Low, _W_Up));
    PP += (P[j] = W[j] * gen_random_range(1.0, _Variance));
    if (W[j] < min_w) min_w = W[j];
  }

  /* generate the total weight bound */
  b_float = ((float) (WW - min_w)) * gen_random_range(0.4, 0.6);
  BOUND = WW - (B = min_w + toint(b_float));

  /* sort P array into sortP of which value is index ordering ascendingly */
  for (i = 1; i <= U; i++) {
    PdivW[i] = tofloat(P[i]) / tofloat(W[i]);
    sortP[i] = i;
  }
  for (i = 1; i < U; i++) {
    best_one = i;
    best = PdivW[sortP[best_one]];
    for (j = i+1; j <= U; j++)
      if (PdivW[sortP[j]] < best) { best_one = j; best = PdivW[sortP[j]]; }
    tmp = sortP[i];
    sortP[i] = sortP[best_one];
    sortP[best_one] = tmp;
  }

#ifdef DEBUG
if (cmd.debug >= 2)
  for (i = 1; i <= U; i++)
    printf("P[%d]=%d, W[%d]=%d\n", i, P[i], i, W[i]);
#endif

  set_node_size(U * sizeof(int));
}

::::::::::::::
solver/ks.pgm/search.c
::::::::::::::

typedef float decomp_;
decomp_ decomp_h_fun ();
decomp_ dh_val[MaxProblemSize];


node_ *expand (node, child_type, nchild)
    node_ *node;
    child_ child_type;
    int nchild;
{   int entity = node->entity;
    domain g_cost = node->g_cost;
    node_ *left_child, *right_child;
    int i, who, gen_left, gen_left_only;
    decomp_ best;

    if (node->ndecomp >= 2) return NULL;  /** infeasible **/

    /** binary decomposition (take or not take) **/
    /** decomposition indicates which vertice should be considered first **/

    if (child_type == NEXT_N_CHILDREN && nchild == 1)
        child_type = NEXT_CHILD;

    /** check whether the left child needs to be generated or not **/
    gen_left_only = (child_type == NEXT_CHILD && node->ndecomp == 0) ? 1 : 0;
    gen_left = (child_type == ALL_CHILDREN || gen_left_only) ? 1 : 0;

    /** init, less penalty is better **/
    for (i = 0; i < U; i++) dh_val[i] = HUGE_FLOAT;

    /** consider all live children **/
    for (i = 0; i < U; i++)
        if (*(node->item+i))
            dh_val[i] = decomp_h_fun (P[i+1], W[i+1], node->depth);

    /** find the best decomposition by decomp heuristic values **/
    best = HUGE_FLOAT;
    who = -1;
    for (i = 0; i < U; i++)
        if (best > dh_val[i]) { best = dh_val[i]; who = i+1; }

    if (who == -1) return NULL;   /** infeasible **/

    if (gen_left)
        left_child = allocate_node (node, who, g_cost+P[who], node->w+W[who]);
    if (gen_left_only) { node->ndecomp = 1; return left_child; }

    right_child = get_search_node_from_pool ();
    init_node_struct (right_child, node);
    right_child->entity = node->entity;
    right_child->g_cost = node->g_cost;
    right_child->w = node->w;
    right_child->lowb = node->lowb;
    right_child->upb = node->upb;
    for (i = 0; i < U; i++) *(right_child->item+i) = *(node->item+i);
    *(right_child->item+who-1) = 0;

    node->ndecomp = 2;
    if (gen_left) {
        left_child->brother = right_child;
        return left_child;
    }
    else return right_child;
}


decomp_ decomp_h_fun (p, w, d)
    int p, w, d;
{
#ifdef HARE_DECOMP
    float DECOMP_H ();
    return (DECOMP_H (p, w, d));
#else
```

```
    return (tofloat (p) / tofloat (w));
#endif
}


int is_feasible (node)
  node_ *node;
{ return (node->w >= BOUND); }


int is_infeasible (node)
  node_ *node;
{ return 0; }
::::::::::::::
solver/ks.pgm/support.c
::::::::::::::

node_ *root_generator ()
{
  /* generate a root node with
   * parent = nil, entity = 0, g_cost = 0, w = 0
   */
  return allocate_node(NULL, 0, 0, 0);
}

/* problem dependent initialization */
void iipd_init ()
{ int i;

  for (i = 0; i < MaxProblemSize; i++) Dom[i] = NULL;
}

/* problem dependent environmental initialization for each iteration */
void idpd_init () { dom_init(); }


/* max sum profit = min sum penalty
 * profit:  items selected for packing constitue profit
 * penalty: items not selected constitute penalty
 * feature: min penalty and the sum of penalty increase from 0
 * all p, w, and cost are positive
 */
node_ *allocate_node (parentp, entity, g_cost, w)
  node_ *parentp;
  int entity, g_cost, w;
{ node_ *p;
  int k;

  p = get_search_node_from_pool();
  init_node_struct(p, parentp);

  p->entity = entity;
  p->g_cost = g_cost;
  p->w = w;

  /* clean these value fields */
  p->lowb = - problem.huge;
  p->upb = problem.huge;

  if (parentp) {
    for (k = 0; k < U; k++) *(p->item+k) = *(parentp->item+k);
    *(p->item+entity-1) = 0;
  } else {
    for(k = 0; k < U; k++) *(p->item+k) = 1;
```

```
  }

  return p;
}


/* allocate associates for newly allocated search nodes */
void pd_region ()
{ int offset, *pool, item_offset, *a, i;
  node_ *p;

  pool = (int *) pool_manager;
  offset = node_conf.node_size / sizeof(int);
  item_offset = sizeof(node_) / sizeof(int);

  for (i = 0; i < AllocationSize; i++) {
    p = ((node_ *) (a = pool + offset * i));

    /* allocate *item frame */
    p->item = a + item_offset;
  }
}


void reset_sol_buf (s)
  solution_ *s;
{ int i;

  for (i = 0; i < problem.size; i++) s->not_in_sack[i] = 0;
}


/* for debug */
void print_conf (fp)
  FILE *fp;
{ int i;

  fprintf(fp, "U=%d, B=%d, BOUND=%d, WW=%d, PP=%d\n", U, B, BOUND, WW, PP);
  for (i = 1; i <= U; i++)
    fprintf(fp, "W[%d]=%d, P[%d]=%d\n", i, W[i], i, P[i]);
}


void evaluate_solution (sp)
  search_ *sp;
{ solution_ *sol;
  int i;

  if (sp) sol = sp->incumbent;
  else error("evaulate_solution: internal error: sp is nil");
  printf("\nSOLUTION:\n");
  for (i = 1; i <= problem.size; i++)
    if (sol->not_in_sack[i] >= 0)
      printf("%d-th object not in sack is  %d\n", i, sol->not_in_sack[i]);
    else break;
}
```

```
Thu Jan 30 15:46:22 CST 1992
::::::::::::::
solver/pp.pgm/define.h
::::::::::::::

#ifndef __PP_define_h_
#define __PP_define_h_

typedef int     domain;
#define Domain  INT

#define AllocationSize          256

#define PDSI_PART       \
        domain  value;          \
        domain  i_value;        /* i_value is sum of (_x - _r) */ \
        domain  *o_values;      /* ptr to array of child's open values */


#define rParamLow       0       /* requirement */
#define rParamUp        3
#define cParamLow       1       /* capacity */
#define cParamUp        4
#define bParamLow       50      /* setup cost */
#define bParamUp        100
#define pParamLow       10      /* production cost */
#define pParamUp        20
#define hParamLow       5       /* inventory cost */
#define hParamUp        10

#define ProductionRange (cParamUp + 2)
#define MaxProblemSize  30

typedef struct
{
        domain  prod[MaxProblemSize];
} solution_;

#endif __PP_define_h_

::::::::::::::
solver/pp.pgm/abc.c
::::::::::::::

domain  rParam[MaxProblemSize]; /* requirement */
domain  cParam[MaxProblemSize]; /* capacity */
domain  bParam[MaxProblemSize]; /* setup cost */
domain  pParam[MaxProblemSize]; /* production cost */
domain  hParam[MaxProblemSize]; /* inventory cost */
domain  xParam[MaxProblemSize]; /* production */
domain  pSort[MaxProblemSize];  /* sorted version of pParam */

::::::::::::::
solver/pp.pgm/bound.c
::::::::::::::

domain pp_eval_lowb (), pp_eval_upb ();


void evaluate_lower_bound (node)
  node_ *node;
{
  node->lowb = is_feasible (node) ? node->g_cost : pp_eval_lowb(node);
}
```

```
domain pp_eval_lowb (node)
  node_ *node;
{ int k = node->entity + 1;
  int m, i, j;
  domain i_value = node->i_value;
  domain cost = node->g_cost;
  domain backlog, diff, x[MaxProblemSize];

  /* reset x array */
  for (i = k; i <= problem.size; i++) x[i] = 0;

  /* backlogging over [k ... i] without considering h effect */
  for (i = k; i <= problem.size; i++)
    if ((backlog = rParam[i] - i_value) < 0) { i_value -= rParam[i]; x[i] = 0; }
    else {
      i_value = 0;
      for (j = 1; j <= problem.size; j++) {
        m = pSort[j];
        if (in(m,k,i))
            if (backlog > (diff = cParam[m] - x[m])) {
              x[m] = cParam[m];
              backlog -= diff;
            }
            else { x[m] += backlog; backlog = 0; break; }
      }

      if (backlog > 0)    /* infeasible solution */ return(problem.huge);
    }

  /* calculate the cost */
  for (i = k; i <= problem.size; i++)
    if (x[i]) cost += (pParam[i] * x[i]);

  return cost;
}


solution_ *evaluate_upper_bound (node, sol)
  node_ *node;
  solution_ *sol;
{
    node->upb = is_feasible (node) ? node->g_cost : pp_eval_upb(node);
  return sol;
}


domain pp_eval_upb (node)
  node_ *node;
{ domain i_value = node->i_value;
  domain cost = node->g_cost;
  domain backlog, diff, x[MaxProblemSize];
  int k = node->entity + 1;
  int i, j;

  /* reset x array */
  for (i = k; i <= problem.size; i++) x[i] = 0;

  /* greedy backlogging over [i ... k]- */
  for (i = k; i <= problem.size; i++)
    if ((backlog = rParam[i] - i_value) < 0) { i_value -= rParam[i]; x[i] = 0; }
    else {
      i_value = 0;
      for (j = i; j >= k; j--)
        if (backlog > (diff = cParam[j] - x[j])) {
          x[j] = cParam[j];
          backlog -= diff;
        }
        else {
          x[j] += backlog;
          backlog = 0;
          break;
        }
      /* infeasible */
      if (backlog > 0) return(problem.huge);
    }

  /* calculate the cost */
  i_value = node->i_value;
  for (i = k; i <= problem.size; i++) {
    i_value += (x[i] - rParam[i]);
    cost += (hParam[i] * i_value);
    if (x[i]) cost += (bParam[i] + pParam[i] * x[i]);
  }
  return cost;
}

::::::::::::::
solver/pp.pgm/gen.c
::::::::::::::

void gen_sample_problem ()
{ domain sum_r = 0, sum_c = 0;
  domain increase, small, temp;
  int i, j;

  for (i = 1; i <= problem.size; i++) {
    sum_r += (rParam[i] = gen_random_int(rParamLow, rParamUp));
    bParam[i] = gen_random_int(bParamLow, bParamUp);
    hParam[i] = gen_random_int(hParamLow, hParamUp);
    sum_c += (cParam[i] = gen_random_int(cParamLow, cParamUp));
    pParam[i] = gen_random_int(pParamLow, pParamUp);
    while (sum_r > sum_c) {
      increase = gen_random_int(cParamLow, (cParamUp + cParamLow) / 2);
      cParam[i] += increase;
      sum_c += increase;
    }
  }

  /* create pSort[] according the ascending order in pParam[]
   * pSort[i] = entity  means  entity has the i-th smallest pParam value.
   */
  /* reset _q */
  for (i = 1; i <= problem.size; i++) pSort[i] = i;

  /* bubble sort ascedingly */
  for (i = 1; i < problem.size; i++) {
    small = i;
    for (j = i + 1; j <= problem.size; j++)
      if (pParam[pSort[small]] > pParam[pSort[j]]) small = j;
      temp = pSort[i]; pSort[i] = pSort[small]; pSort[small] = temp;
  }

  set_node_size(ProductionRange * sizeof(domain));

#ifdef DEBUG
if (cmd.debug >= 2) print_conf(stdout);
#endif
}

::::::::::::::
solver/pp.pgm/search.c
::::::::::::::

typedef float decomp_;
decomp_ decomp_h_fun ();
decomp_ dh_val[ProductionRange];
node_ *ch_arr[ProductionRange];


node_ *expand (node, child_type, nchild)
  node_ *node;
  child_ child_type;
  int nchild;
{ int entity = node->entity;
  int next_entity = node->entity+1;
  domain g_cost = node->g_cost;
  domain i_value = node->i_value;
  domain *o_vals = node->o_values;
  node_ *new_list = NULL;
  node_ *child, *next_child;
  int i, count = 0, who;
  domain w, i_val, cost, val, *ovp;
  decomp_ best;

  switch (child_type) {
  case ALL_CHILDREN: nchild = HUGE_INT; break;
  case NEXT_CHILD:   nchild = 1; break;
  default: break;
  }

  /** init **/
  for (i = 0; i < ProductionRange; i++) {
    dh_val[i] = HUGE_FLOAT;
    ch_arr[i] = NULL;
  }

  /** sprout all necessary children **/
  for (val = *o_vals; val >= 0; val = *(++o_vals)) {
    i_val = i_value + (val - rParam[next_entity]);
    cost = g_cost + hParam[next_entity] * i_val;
    if (val) cost += (bParam[next_entity] + val * pParam[next_entity]);
    ch_arr[val] = allocate_node (node, next_entity, val, i_val, cost);
    gen_open_value (ch_arr[val]);
    dh_val[val] = decomp_h_fun (val, i_val, next_entity);
    if (++count >= nchild) break;
  }

  /** update open values, where 'node->o_values' won't change **/
  if (val < 0) *(node->o_values) = -1;
  else {
    ovp = ++o_vals;  /** new 1st open value **/
    o_vals = node->o_values;
    while (*ovp >= 0) *(o_vals++) = *(ovp++);
    *o_vals = -1;
  }

  /** sort all these children by decomp heuristic values **/
  while (count-- > 0) {
    best = HUGE_FLOAT;
    who = -1;
    for (val = 0; val < ProductionRange; val++)
      if (best > dh_val[val]) { best = dh_val[val]; who = val; }
    ch_arr[who]->next = new_list;
    new_list = ch_arr[who];
    dh_val[who] = HUGE_FLOAT;
  }

  /** reverse new_list **/
  next_child = new_list;
  new_list = NULL;
  while ((child = next_child)) {
    next_child = child->next;
    child->next = NULL;
    child->brother = new_list;
    new_list = child;
  }

  return new_list;
}


decomp_ decomp_h_fun (val, i_val, next_entity)
  int val, i_val, next_entity;
{ decomp_ retval;

  retval = (pParam[next_entity] + hParam[next_entity]) * val +
           (val ? 1 : 0) * bParam[next_entity] +
           hParam[next_entity] * (i_val - rParam[next_entity]);

  return retval;
}


gen_open_value (node)
  node_ *node;
{ int k = node->entity + 1, i;
  domain from, to, val, size;

  /* set lower bound to open values */
  if ((from = rParam[k] - node->i_value) < 0) from = 0;

  /* set upper bound to and size of open values */
  size = (to = cParam[k]) - from + 2;

  if (size > 0) {
    /* initialize the space */
    for (i = 0, val = to; val >= from; val--, i++) *(node->o_values + i) = val;
    *(node->o_values + size - 1) = -1;
  }
  else *(node->o_values) = -1;
}


int is_feasible(node) node_ *node; { return (node->entity == problem.size); }
int is_infeasible(node) node_ *node; { return 0; }
int is_dominated(node) node_ *node; { return 0; }

::::::::::::::
solver/pp.pgm/support.c
::::::::::::::

node_ *root_generator ()
{ node_ *root;

  /* generate a root node */
  root = allocate_node(NULL, 0, 0, 0, 0);
  gen_open_value(root);
  return root;
}

void iipd_init () {}
void idpd_init () {}


node_ *allocate_node (parentp, entity, value, i_value, g_cost)
  node_ *parentp;
  int entity;
  domain value, i_value, g_cost;
{ node_ *p;

  p = get_search_node_from_pool();
  init_node_struct(p, parentp);

  p->entity = entity;
  p->g_cost = g_cost;
  p->value = value;
  p->i_value = i_value;

  /* clean up these value fields */
  p->lowb = - problem.huge;
  p->upb = problem.huge;

  return p;
}


void reset_sol_buf (s)
  solution_ *s;
{ int i;

  for (i = 0; i < problem.size; i++) s->prod[i] = 0;
}


void print_conf (fp)
  FILE *fp;
{ int i;

  for (i = 1; i <= problem.size; i++) {
    fprintf(fp, "x[%d]=%d, r[%d]=%d, c[%d]=%d, p[%d]=%d, ",
            i, xParam[i], i, rParam[i], i, cParam[i], i, pParam[i]);
    fprintf(fp, "h[%d]=%d, b[%d]=%d\n", i, hParam[i], i, bParam[i]);
  }
}


void evaluate_solution (sp)
  search_ *sp;
{ solution_ *sol;
  int i;

  if (sp) sol = sp->incumbent;
  else error("evaulate_solution: internal error: sp is nil");
  printf("\nSOLUTION:\n");
  for (i = 0; i < problem.size; i++)
    printf("%d-th production is %d\n", i, sol->prod[i]);
}
```

```
/* allocate associates for newly allocated search nodes */
void pd_region()
{ int offset, *pool, o_val_offset, *a, i;
  node_ *p;

  pool = (int *) pool_manager;
  offset = node_conf.node_size / sizeof(int);
  o_val_offset = sizeof(node_) / sizeof(int);

  for(i = 0; i < AllocationSize; i++) {
    p = ((node_ *) (a = pool + offset * i));
    p->o_values = a + o_val_offset;     /* allocate *o_values frame */
  }
}
```

```
Thu Jan 30 15:46:59 CST 1992
::::::::::::::
solver/vc.pgm/define.h
::::::::::::::
#ifndef __VC_define_h__
#define __VC_define_h__

typedef int     domain;
#define Domain  INT

#define AllocationSize          256

#define PDSI_PART \
        int     num_edges;      /** number of edges covered **/  \
        int     *vertattr;      /** vertex attributes **/

#define MaxProblemSize           250
#define MaxNumberEdges           (MaxProblemSize * (MaxProblemSize - 1) / 2)

typedef struct { int vertices[MaxProblemSize]; } solution_;
typedef struct { int x, y; } edge;

#define VC_ALIVE        0
#define VC_TAKEN        -1
#define VC_DEAD         -2

#define is_vertex_alive(node,v)         (*(node->vertattr + v) >= 0)
#define is_vertex_taken(node,v)         (*(node->vertattr + v) == VC_TAKEN)
#define is_vertex_dead(node,v)          (*(node->vertattr + v) == VC_DEAD)

#define set_vertex_taken(node,v)        *(node->vertattr + v) = VC_TAKEN
#define set_vertex_dead(node,v)         *(node->vertattr + v) = VC_DEAD

#define set_vertex_degree(node,v,deg)   *(node->vertattr + v) = deg
#define get_vertex_degree(node,v)       (*(node->vertattr + v))


extern int      NumVertices, NumEdges;
extern float    Threshold;
extern int      Adjacency[MaxProblemSize][MaxProblemSize];
extern edge     Edges[MaxNumberEdges];

#endif

::::::::::::::
solver/vc.pgm/bound.c
::::::::::::::

domain vc_eval_lowb (), vc_eval_upb ();

void evaluate_lower_bound (node)
    node_ *node;
{
    node->lowb = is_feasible (node) ? node->g_cost : vc_eval_lowb (node);
}

int sort_vertices[MaxProblemSize];

domain vc_eval_lowb (node)
    node_ *node;
{   int num_edges = node->num_edges;
    domain lowb = node->g_cost;
    int neleft = NumEdges - node->num_edges;    /** num of edges uncovered **/
    int sum, nalive, who, best, temp, i, j;

    /** record who are alive **/
    for (nalive = i = 0; i < NumVertices; i++)
        if(is_vertex_alive (node, i)) sort_vertices[nalive++] = i;

    if (nalive <= 0) return problem.huge;   /** infeasible **/

    /** sort these alive **/
    for (i = 0; i < nalive-1; i++) {
        who = i;
        best = get_vertex_degree (node, sort_vertices[who]);
        for (j = i+1; j < nalive; j++)
            if (best < get_vertex_degree (node, sort_vertices[j])) {
                /** update who and best **/
                who = j;
                best = get_vertex_degree (node, sort_vertices[j]);
            }
        /** swap i and who **/
        temp = sort_vertices[i];
        sort_vertices[i] = sort_vertices[who];
        sort_vertices[who] = temp;
    }

    /** calc lower bound **/
    for (sum = i = 0; i < nalive; i++) {
        lowb++;
        sum += get_vertex_degree (node, sort_vertices[i]);
        if (sum >= neleft) {
            if (node->parent)
                if (node->parent->lowb > lowb) lowb = node->parent->lowb;
            return lowb;
        }
    }

    return problem.huge;
}


solution_ *evaluate_upper_bound (node, sol)
    node_ *node;
    solution_ *sol;
{
    node->upb = vc_eval_upb (node, sol);
    return sol;
}


int check_edges[MaxNumberEdges];

domain vc_eval_upb (node, sol)
    node_ *node;
    solution_ *sol;
{   domain upb = node->g_cost;
    int num_edges = 0;
    int sum, nalive, who, best, temp, i, j;

    /** record who are alive **/
    for (nalive = i = 0; i < NumVertices; i++)
        if(is_vertex_alive (node, i)) sort_vertices[nalive++] = i;

    if (nalive <= 0) return problem.huge;  /** infeasible **/

    /** sort these alive **/
    for (i = 0; i < nalive-1; i++) {
        who = i;
        best = get_vertex_degree (node, sort_vertices[who]);
        for (j = i+1; j < nalive; j++)
            if (best < get_vertex_degree (node, sort_vertices[j])) {
                /** update who and best **/
                who = j;
                best = get_vertex_degree (node, sort_vertices[j]);
            }
        /** swap i and who **/
        temp = sort_vertices[i];
        sort_vertices[i] = sort_vertices[who];
        sort_vertices[who] = temp;
    }

    /** clear up edge-checking array **/
    for (i = 0; i < NumEdges; i++) check_edges[i] = VC_ALIVE;

    /** fill in edge-checking array **/
    for (i = 0; i < NumVertices; i++) {
        if (is_vertex_taken (node, i)) {
            sol->vertices[i] = 1;
            num_edges += mark_edges (i, check_edges, VC_TAKEN);
        }
    }

    for (i = 0; i < nalive; i++) {
        if (num_edges < NumEdges) {
            upb++;
            sol->vertices[sort_vertices[i]] = 1;
            num_edges += mark_edges (sort_vertices[i], check_edges, VC_TAKEN);
        }
        else break;
    }

    return ((num_edges >= NumEdges) ? upb : problem.huge);
}


mark_edges (i, arr, mark)
    int i;
    int arr[];
    int mark;
{   int j, count = 0;

    for (j = 0; j < NumVertices; j++)
        if (Adjacency[i][j] >= 0)
            if (arr[Adjacency[i][j]] == VC_ALIVE) {
                arr[Adjacency[i][j]] = mark;
                count++;
            }

    return count;
}
```

```
::::::::::::::
solver/vc.pgm/define.c
::::::::::::::

int     NumVertices;    /** num of vertices in graph **/
int     NumEdges;       /** num of edges in graph **/
float   Threshold;      /** threshold for sprouting edges **/

int     Adjacency[MaxProblemSize][MaxProblemSize];
edge    Edges[MaxNumberEdges];
```

```
::::::::::::::
solver/vc.pgm/gen.c
::::::::::::::

void gen_sample_problem ()
{   int i, j;
    FILE *fp, *fopen ();

    NumVertices = problem.size;
    NumEdges = 0;

    /** clean up problem attributes **/
    for (i = 0; i < NumVertices; i++)
        for (j = 0; j < NumVertices; j++)
            Adjacency[i][j] = VC_DEAD;

    /** randomly generate connectivity (0.1, 0.2) **/
    /** Threshold = 3.0 / problem.size; **/
    Threshold = 0.1;
    /** randomly generate edges according to Threshold **/
    for (i = 0; i < NumVertices-1; i++)
        for (j = i; j < NumVertices; j++)
            if (i != j)
                if (gen_random_float () <= Threshold) {
                    Adjacency[i][j] = Adjacency[j][i] = NumEdges;
                    Edges[NumEdges].x = i;
                    Edges[NumEdges].y = j;
                    NumEdges++;
                }

  set_node_size (NumVertices * sizeof (int));

  fp = fopen (".vc", "a");
  fprintf (fp, "VC(%d,%d,%d)\n", problem.size, NumEdges, problem.seed);
  fclose (fp);

}
```

```
::::::::::::::
solver/vc.pgm/search.c
::::::::::::::

typedef int decomp_;
decomp_ decomp_h_fun ();
decomp_ dh_val[MaxProblemSize];


node_ *expand (node, child_type, nchild)
    node_ *node;
    child_ child_type;
    int nchild;
{   int entity = node->entity;
    domain g_cost = node->g_cost;
    node_ *left_child, *right_child;
    int i, who, gen_left, gen_left_only;
    decomp_ best;

    if (node->ndecomp >= 2) return NULL;  /** infeasible **/

    /** binary decomposition (take or not take) **/
    /** decomposition indicates which vertice should be considered first **/

    if (child_type == NEXT_N_CHILDREN && nchild == 1)
```

```
        child_type = NEXT_CHILD;

    /** check whether the left child needs to be generated or not **/
    gen_left_only = (child_type == NEXT_CHILD && node->ndecomp == 0) ? 1 : 0;
    gen_left = (child_type == ALL_CHILDREN || gen_left_only) ? 1 : 0;

    /** init **/
    for (i = 0; i < NumVertices; i++) dh_val[i] = 0;

    /** consider all live children **/
    for (i = 0; i < NumVertices; i++)
        if (is_vertex_alive (node, i)) dh_val[i] = decomp_h_fun (node, i);

    /** find the best decomposition by decomp heuristic values **/
    best = 0;
    who = -1;
    for (i = 0; i < NumVertices; i++)
        if (best < dh_val[i]) { best = dh_val[i]; who = i; }

    if (who == -1) return NULL;   /** infeasible **/

    if (gen_left) left_child = allocate_node (node, who, g_cost+1);
    if (gen_left_only) { node->ndecomp = 1; return left_child; }

    right_child = get_search_node_from_pool ();
    init_node_struct (right_child, node);
    right_child->entity = node->entity;
    right_child->g_cost = node->g_cost;
    right_child->num_edges = node->num_edges;
    right_child->lowb = node->lowb;
    right_child->upb = node->upb;
    for (i = 0; i < NumVertices; i++)
        *(right_child->vertattr+i) = *(node->vertattr+i);
    set_vertex_dead (right_child, who);

    node->ndecomp = 2;
    if (gen_left) {
        left_child->brother = right_child;
        return left_child;
    }
    else return right_child;
}


decomp_ decomp_h_fun (node, i)
  node_ *node;
  int i;
{
  return get_vertex_degree (node, i);
}


int is_feasible (node)
    node_ *node;
{ return ((node->num_edges >= NumEdges) ? 1 : 0); }


int is_infeasible (node)
    node_ *node;
{ return 0; }

::::::::::::::
solver/vc.pgm/support.c
::::::::::::::
```

```
node_ *root_generator ()
{   /** generate a root node with
     ** parent = nil, entity = -1, g_cost = 0 **/
    return (allocate_node (NULL, -1, (domain) 0));
}


void iipd_init () {}
void idpd_init () {}
int is_dominated (node) node_ *node; { return 0; }


node_ *allocate_node (parentp, entity, g_cost)
    node_ *parentp;
    int entity;
    domain g_cost;
{   node_ *p;
    int i, j;
    int check_edge[MaxNumberEdges];

    p = get_search_node_from_pool ();
    init_node_struct (p, parentp);

    p->entity = entity;
    p->g_cost = g_cost;

    /** clean these value fields **/
    p->lowb = - problem.huge;
    p->upb = problem.huge;

    /** troublesome to calc live degree of vertex **/

    /** collect info about which edges are alive **/
    for (i = 0; i < NumEdges; i++) check_edge[i] = VC_ALIVE;
    for (i = 0; i < NumVertices; i++) *(p->vertattr+i) = 0;
    if (parentp)
        for (i = 0; i < NumVertices; i++)
            if (i == entity || is_vertex_taken (parentp, i))
                for (j = 0; j < NumVertices; j++)
                    if (Adjacency[i][j] >= 0)
                        check_edge[Adjacency[i][j]] = VC_TAKEN;

    /** calc live degree of all vertices **/
    p->num_edges = NumEdges;
    for (i = 0; i < NumEdges; i++)
        if (check_edge[i] == VC_ALIVE) {
            --(p->num_edges);
            (*(p->vertattr + Edges[i].x))++;
            (*(p->vertattr + Edges[i].y))++;
        }

    /** disable those vertices which are taken or dead **/
    if (parentp) {
        for (i = 0; i < NumVertices; i++) {
            if (is_vertex_taken (parentp, i)) set_vertex_taken (p, i);
            else if (is_vertex_dead (parentp, i)) set_vertex_dead (p, i);
        }
        set_vertex_taken (p, entity);
    }

    return p;
}
```

```
void pd_region ()
/** allocate associates for newly allocated search nodes **/
{   int offset, *pool, vertattr_offset, *a, i;
    node_ *p;

    pool = (int *) pool_manager;
    offset = node_conf.node_size / sizeof (int);
    vertattr_offset = sizeof (node_) / sizeof (int);

    for (i = 0; i < AllocationSize; i++) {
        p = ((node_ *) (a = pool + offset * i));
        /** allocate *vertattr frame **/
        p->vertattr = a + vertattr_offset;
    }
}


void reset_sol_buf (s)
    solution_ *s;
{   int i;

    for (i = 0; i < problem.size; i++) s->vertices[i] = 0;
}


void print_conf (fp)
    FILE *fp;
{   int i, j;

    fprintf (fp, "NumVertices=%d, NumEdges=%d, Threshold=%f\n",
        NumVertices, NumEdges, Threshold);
    fprintf (fp, "Adjacency\n");
    for (i = 0; i < NumVertices; i++) {
        for (j = 0; j < NumVertices; j++)
            fprintf (fp, "%d, ", Adjacency[i][j]);
        fprintf (fp, "\n");
    }
    fprintf (fp, "Edges\n");
    for (i = 0; i < NumEdges; i++)
        fprintf (fp, "E[%d]: x=%d, y=%d\n", i, Edges[i].x, Edges[i].y);
}


void evaluate_solution (sp)
    search_ *sp;
{   solution_ *sol;
    int i, count = 0;

#ifdef DEBUG
if (cmd.debug >= 5)
    print_conf (stdout);
#endif
    if (sp) sol = sp->incumbent;
    else error ("evaulate_solution: internal error: sp is nil");
    printf ("\nSOLUTION:\n");
    for (i = 0; i < problem.size; i++)
        if (sol->vertices[i]) {
            printf ("%d-th vertex is selected\n", i);
            count++;
        }
    printf ("solution value = %d\n", count);
}
```

```
Thu Jan 30 15:47:13 CST 1992
::::::::::::::
solver/wct.pgm/abc.c
::::::::::::::

#define MPS     MaxProblemSize

int     nTasks;         /* # tasks */
int     nProcessors;    /* # processors */

domain  execTime[MPS];  /** execution times of tasks**/
float   Weight[MPS];    /** weights of tasks **/

int     sortTask[MPS];  /** sorted version of tasks **/
float   WdivT[MPS];     /** weights div times **/

#undef MR

::::::::::::::
solver/wct.pgm/bound.c
::::::::::::::

domain grcs_eval_lowb (), grcs_eval_upb ();
domain p_ck[MaxProblemSize];


void evaluate_lower_bound (node)
  node_ *node;
{
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval lowb\n");
#endif

  node->lowb = is_feasible (node) ? node->g_cost : grcs_eval_lowb (node);
}


domain grcs_eval_lowb (node)
  node_ *node;
{ domain lowb = node->g_cost;
  domain min_t = huge_float;
  float min_w = huge_float;
  int i, j, who;
  domain alloc_proc ();

  for (i = 0; i < nProcessors; i++) p_ck[i] = *(node->proc_ck+i);

  for (i = 0; i < nTasks; i++)
    if (is_task_alive (node, i))
      if (min_w > Weight[i]) min_w = Weight[i];

  for (i = 0; i < nTasks; i++)
    if (is_task_alive (node, i)) {
      min_t = alloc_proc (p_ck, &who);
      p_ck[who] += execTime[i];
      lowb += (p_ck[who] * min_w);
    }

  return lowb;
}


solution_ *evaluate_upper_bound (node, sol)
  node_ *node;
  solution_ *sol;
{
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval upb\n");
#endif

  node->upb = is_feasible (node) ? node->g_cost : grcs_eval_upb (node, sol);

  return sol;
}


domain grcs_eval_upb (node, sol)
  node_ *node;
  solution_ *sol;
{ domain upb = node->g_cost;
  domain min_t = huge_float;
  float min_w = huge_float;
  int i, j, who;
  domain alloc_proc ();

  for (i = 0; i < nProcessors; i++) p_ck[i] = *(node->proc_ck+i);

  for (i = 0; i < nTasks; i++)
    if (is_task_sched (node, i)) sol->schedTime[i] = get_task_time (node, i);

  for (i = 0; i < nTasks; i++) {
    j = sortTask[i];
    if (is_task_alive (node, j)) {
      min_t = alloc_proc (p_ck, &who);
      sol->schedTime[j] = min_t;
      p_ck[who] += execTime[j];
      upb += (p_ck[who] * Weight[j]);
    }
  }

  return upb;
}


domain alloc_proc (p_ck_arr, whop)
  domain p_ck_arr[];
  int *whop;
{ domain tmin;
  int i;

  *whop = 0;
  tmin = p_ck_arr[0];
  for (i = 1; i < nProcessors; i++)
    if (tmin > p_ck_arr[i]) {
      tmin = p_ck_arr[i];
      *whop = i;
    }

  return tmin;
}

::::::::::::::
solver/wct.pgm/config.h
::::::::::::::

#ifndef __RCS_config_h_
#define __RCS_config_h_
```

```
/* bound.c */
extern solution_ *evaluate_upper_bound();
extern void      evaluate_lower_bound(), update_bounds();

/* gen.c */
extern void      gen_sample_problem();

/* search.c */
extern int       is_infeasible(), is_feasible();
extern node_     *expand();

/* support.c */
extern int       is_dominated();
extern node_     *root_generator(), *allocate_node();
extern domain    get_problem_domain();
extern void      iipd_init(), idpd_init(), pd_region(), reset_sol_buf(),
                 print_conf();

#endif  __RCS_config_h_

::::::::::::::
solver/wct.pgm/define.h
::::::::::::::
#ifndef __WCT_define_h_
#define __WCT_define_h_

typedef float   domain;
#define Domain  FLOAT

#define MaxProblemSize  30
#define AllocationSize  256

#define PDSI_PART         \
        domain  *proc_ck;       /** processor's clocks **/          \
        int     *open_task;     /** open pointer, to be selected **/    \
        float   *sched_time;    /** scheduled times of tasks **/


typedef struct
{
        domain  schedTime[MaxProblemSize];
} solution_;


#define NOT_SCHED       (-1.0)

#define is_task_alive(node,i)    (*(node->sched_time + i) < 0.0)
#define is_task_sched(node,i)    (*(node->sched_time + i) >= 0.0)
#define set_task_alive(node,i)   *(node->sched_time + i) = NOT_SCHED
#define get_task_time(node,i)    (*(node->sched_time + i))
#define set_task_time(node,i,t)  *(node->sched_time + i) = t

#define _T_Low          10.0            /** exec time **/
#define _T_Up           100.0

#define MaxProcessor    8

#endif  __WCT_define_h_

::::::::::::::
solver/wct.pgm/gen.c
::::::::::::::

void gen_sample_problem()
{ int i, j, best_one, temp;
  float max_ratio = 0.0;

  /* generate the numbers of resources randomly */
  nTasks = problem.size;
  nProcessors = 3;

  /** find out the sort version of tasks **/
  for (i = 0; i < nTasks; i++) {
    sortTask[i] = i;
    execTime[i] = gen_random_range (_T_Low, _T_Up);
    WdivT[i] = gen_random_range (0.9, 1.1);
    Weight[i] = WdivT[i] * execTime[i];
  }
  for (i = 0; i < nTasks; i++) {
    best_one = i;
    max_ratio = WdivT[sortTask[best_one]];
    for (j = i+1; j < nTasks; j++)
      if (WdivT[sortTask[j]] > max_ratio) {
        best_one = j;
        max_ratio = WdivT[sortTask[j]];
      }
    temp = sortTask[i];
    sortTask[i] = sortTask[best_one];
    sortTask[best_one] = temp;
  }

  set_node_size ((nProcessors + nTasks) * sizeof (float) +
                 nTasks * sizeof (int));
}

::::::::::::::
solver/wct.pgm/output.h
::::::::::::::
#ifndef __TS_output_h_
#define __TS_output_h_

/*
        Output Format Control Signals
        Every control signals must be defined to be either 1 or 0.
*/

#define         OUT_GRAPH               1
#define         OUT_SUMMARY             1

#define         OUT_TIME_LIMIT          1
#define         OUT_SPACE_LIMIT         0
#define         OUT_CST_LIMIT           0

#define         OUT_REAL_TIME           0
#define         OUT_REAL_MAX_SPACE      0
#define         OUT_REAL_CST            0

#define         OUT_VIRTUAL_TIME        1
#define         OUT_VIRTUAL_MAX_SPACE   1
#define         OUT_VIRTUAL_CST         1

#define         OUT_ROOT_APPROX         1
#define         OUT_RUN_TIME_APPROX     1
#define         OUT_APPROX              1

#define         OUT_INCUMBENT           1
#define         OUT_LOWB                1
#define         OUT_THRESHOLD           1
```

```
/*      sTCA & sTCGD & pTCGD            */
#define         OUT_GRADIENT_FACTOR     1

/*      pTCA & dTCA & pTCGD     */
#define         OUT_STOPPING_FACTOR     1
#define         OUT_CORRECTIVE_FACTOR   1

#endif __TS_output_h_

::::::::::::::
solver/wct.pgm/search.c
::::::::::::::

typedef float decomp_;
decomp_ decomp_h_fun ();
decomp_ dh_val[MaxProblemSize];
node_ *ch_arr[MaxProblemSize];


node_ *expand (node, child_type, nchild)
  node_ *node;
  child_ child_type;
  int nchild;
{ int entity = node->entity;
  domain g_cost = node->g_cost;
  node_ *new_list = NULL;
  node_ *child, *next_child;
  int i, count = 0, who;
  domain w;
  decomp_ best;

  switch (child_type) {
  case ALL_CHILDREN:  nchild = HUGE_INT;
  case NEXT_N_CHILDREN:

    /** init **/
    for (i = 0; i < nTasks; i++) {
      dh_val[i] = (decomp_) 0;;
      ch_arr[i] = NULL;
    }

    /** sprout all necessary children **/
    for (i = 0; i < nTasks; i++)
      if (*(node->open_task+i)) {
        *(node->open_task+i) = 0;
        ch_arr[i] = allocate_node (node, i);
        dh_val[i] = decomp_h_fun (i);
        if (++count >= nchild) break;
      }

    /** sort all these children by decomp heuristic values **/
    while (count-- > 0) {
      best = (decomp_) 0;
      who = -1;
      for (i = 0; i < nTasks; i++)
        if (best < dh_val[i]) { best = dh_val[i]; who = i; }
      ch_arr[who]->next = new_list;
      new_list = ch_arr[who];
      dh_val[who] = (decomp_) 0;
    }

    /** reverse new_list **/
    next_child = new_list;
```

```
    new_list = NULL;
    while ((child = next_child)) {
      next_child = child->next;
      child->next = NULL;
      child->brother = new_list;
      new_list = child;
    }

    return new_list;
    break;

  case NEXT_CHILD:
    for (i = 0; i < nTasks; i++)
      if (*(node->open_task+i)) {
        *(node->open_task+i) = 0;
        return allocate_node (node, i);
      }
    break;

  default: break;
  }

  return NULL;
}


decomp_ decomp_h_fun (entity)
  int entity;
{
  return ((decomp_) execTime[entity]);
}


int is_infeasible (node)
  node_ *node;
{ return 0; }


int is_feasible (node)
  node_ *node;
{ int i;

  for (i = 0; i < nTasks; i++)
    if (is_task_alive (node, i)) return 0;

  return 1;
}

::::::::::::::
solver/wct.pgm/support.c
::::::::::::::

node_ *root_generator () {
  return allocate_node (NULL, -1);
}

void iipd_init() { }

void idpd_init() { }

int is_dominated (node) node_ *node; { return 0; }


node_ *allocate_node (parentp, entity)
```

```
  node_ *parentp;
  int entity;
{ node_ *p;
  int i, j, who;
  domain min_t;

  p = get_search_node_from_pool();
  init_node_struct(p, parentp);

  p->entity = entity;
  p->g_cost = (domain) 0.0;

  /* clean thses value fields */
  p->lowb = - problem.huge;
  p->upb = problem.huge;

  if (parentp)
    for (i = 0; i < nTasks; i++) *(p->sched_time+i) = *(parentp->sched_time+i);
  else
    for (i = 0; i < nTasks; i++) set_task_alive (p, i);

  if (parentp) {
    for (i = 0; i < nProcessors; i++)
      *(p->proc_ck+i) = *(parentp->proc_ck+i);
  } else {
    for (i = 0; i < nProcessors; i++)
      *(p->proc_ck+i) = (domain) 0.0;
  }

  if (parentp) {
    min_t = alloc_proc (p->proc_ck, &who);
    *(p->sched_time+entity) = min_t;
    min_t += execTime[entity];
    *(p->proc_ck+who) = min_t;
    p->g_cost = parentp->g_cost + min_t * Weight[entity];
  }

  for (i = 0; i < nTasks; i++)
    if (is_task_alive (p, i)) *(p->open_task+i) = 1;

  return p;
}


/* allocate associates for newly allocated search nodes */
void pd_region ()
{ int offset, *pool, pck_offset, open_offset, sched_time_offset;
  int *a, i;
  node_ *p;

  pool = (int *) pool_manager;
  offset = node_conf.node_size / sizeof(int);
  pck_offset = sizeof(node_) / sizeof(int);
  open_offset = pck_offset + nProcessors * sizeof (domain) / sizeof (int);
  sched_time_offset = open_offset + nTasks;

  for (i = 0; i < AllocationSize; i++) {
    p = ((node_ *) (a = pool + offset * i));
    p->proc_ck = (domain *) (a + pck_offset);
    p->open_task = (int *) (a + open_offset);
    p->sched_time = (domain *) (a + sched_time_offset);
  }
}
```

```
void reset_sol_buf (s)
  solution_ *s;
{ int i;

  for (i = 0; i < problem.size; i++) s->schedTime[i] = (domain) 0.0;
}


void evaluate_solution (sp)
  search_ *sp;
{ solution_ *sol;
  int i, j;

  if (sp) sol = sp->incumbent;
  else error("evaulate_solution: internal error: sp is nil");

  printf ("\nPROBLEM:\n");
  printf ("nTasks=%d, nProcessors=%d\n", nTasks, nProcessors);

  for (i = 0; i < nTasks; i++)
    printf ("execTime[%d]=%g, Weight[%d]=%g\n",
            i, (float) execTime[i], i, (float) Weight[i]);

  printf("\nSOLUTION:\n");
  for (i = 0; i < problem.size; i++)
    printf("%d-th task is scheduled at time  %g\n",
            i, (float) (sol->schedTime[i]));
}
```

```
Thu Jan 30 15:47:05 CST 1992
::::::::::::::
solver/grcs.pgm/abc.c
::::::::::::::

#define MPS     MaxProblemSize
#define MR      MaxResource

int     nTasks;         /* # tasks */
int     nProcessors;    /* # processors */
int     nResources;     /* # resources */
int     REQ[MPS][MR];   /* resource requirements */

domain  execTime[MPS];  /** execution times of tasks**/

#undef MPS
#undef MR

::::::::::::::
solver/grcs.pgm/bound.c
::::::::::::::

domain grcs_eval_lowb (), grcs_eval_upb ();


void evaluate_lower_bound (node)
  node_ *node;
{
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval lowb\n");
#endif

  node->lowb = is_feasible (node) ? node->g_cost : grcs_eval_lowb (node);
}


domain grcs_eval_lowb (node)
  node_ *node;
{ domain lowb = node->g_cost;
  domain texec = (domain) 0.0;
  domain tavail = (domain) 0.0;
  domain tmax = (domain) 0.0;
  int i, j, who;

  /** calc processors' bottleneck **/

  for (i = 0; i < nTasks; i++)
    if (is_task_alive (node, i)) texec += execTime[i];

  for (i = 0; i < nProcessors; i++)
    if (tmax < *(node->proc_ck+i)) tmax = *(node->proc_ck+i);

  tavail = (domain) 0.0;
  for (i = 0; i < nProcessors; i++) tavail += tmax - *(node->proc_ck+i);

  texec -= tavail;
  if (texec > (domain) 0.0) lowb += (texec / nProcessors);

  /** calc resources' bottlenecks **/
  for (j = 0; j < nResources; j++) {
    texec = (domain) 0.0;
    for (i = 0; i < nTasks; i++)
      if (is_task_alive (node, i) && REQ[i][j]) texec += execTime[i];
    if (lowb < texec) lowb = texec;
```

```
  }

  if (node->parent)
    if (lowb < node->parent->lowb) lowb = node->parent->lowb;

  return lowb;
}


solution_ *evaluate_upper_bound (node, sol)
  node_ *node;
  solution_ *sol;
{
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval upb\n");
#endif

  node->upb = is_feasible (node) ? node->g_cost : grcs_eval_upb (node, sol);

  return sol;
}


domain r_ck[MaxProblemSize], p_ck[MaxProblemSize];
domain grcs_eval_upb (node, sol)
  node_ *node;
  solution_ *sol;
{ domain tmax = (domain) 0.0;
  int i, j, who;
  domain alloc_proc ();

  for (i = 0; i < nResources; i++) r_ck[i] = *(node->resource_ck+i);
  for (i = 0; i < nProcessors; i++) p_ck[i] = *(node->proc_ck+i);

  for (i = 0; i < nTasks; i++)
    if (is_task_sched (node, i)) sol->schedTime[i] = get_task_time (node, i);

  for (i = 0; i < nTasks; i++) {
    if (is_task_alive (node, i)) {

      tmax = alloc_proc (p_ck, &who);

      for (j = 0; j < nResources; j++)
        if (REQ[i][j])
          /** resource j is required **/
          if (tmax < r_ck[j]) tmax = r_ck[j];

      sol->schedTime[i] = tmax;
      tmax += execTime[i];
      p_ck[who] = tmax;

      for (j = 0; j < nResources; j++)
        if (REQ[i][j]) r_ck[j] = tmax;
    }
  }

  tmax = (domain) 0.0;
  for (i = 0; i < nProcessors; i++)
    if (tmax < p_ck[i]) tmax = p_ck[i];

  return tmax;
}
```

```
domain alloc_proc (p_ck_arr, whop)
  domain p_ck_arr[];
  int *whop;
{ domain tmin;
  int i;

  *whop = 0;
  tmin = p_ck_arr[0];
  for (i = 1; i < nProcessors; i++)
    if (tmin > p_ck_arr[i]) {
      tmin = p_ck_arr[i];
      *whop = i;
    }

  return tmin;
}

:::::::::::::
solver/grcs.pgm/config.h
:::::::::::::

#ifndef __RCS_config_h_
#define __RCS_config_h_

/* bound.c */
extern solution_ *evaluate_upper_bound();
extern void      evaluate_lower_bound(), update_bounds();

/* gen.c */
extern void      gen_sample_problem();

/* search.c */
extern int       is_infeasible(), is_feasible();
extern node_     *expand();

/* support.c */
extern int       is_dominated();
extern node_     *root_generator(), *allocate_node();
extern domain    get_problem_domain();
extern void      iipd_init(), idpd_init(), pd_region(), reset_sol_buf(),
                 print_conf();

#endif  __RCS_config_h_

:::::::::::::
solver/grcs.pgm/define.h
:::::::::::::
#ifndef __GRCS_define_h_
#define __GRCS_define_h_

typedef float   domain;
#define Domain  FLOAT

#define MaxProblemSize  160
#define AllocationSize  256

#define PDSI_PART       \
        domain  *proc_ck;       /** processor's clocks **/              \
        domain  *resource_ck;   /** resource's clocks **/               \
        int     *open_task;     /** open pointer, to be selected **/    \
        float   *sched_time;    /** scheduled times of tasks **/


typedef struct
{
        domain  schedTime[MaxProblemSize];
} solution_;


#define NOT_SCHED       (-1.0)

#define is_task_alive(node,i)    (*(node->sched_time + i) < 0.0)
#define is_task_sched(node,i)    (*(node->sched_time + i) >= 0.0)
#define set_task_alive(node,i)   *(node->sched_time + i) = NOT_SCHED
#define get_task_time(node,i)    (*(node->sched_time + i))
#define set_task_time(node,i,t)  *(node->sched_time + i) = t



#define _n_R_Low        4               /* num of resources */
#define _n_R_Up         4
#define _n_P_Low        3               /* num of processors */
#define _n_P_Up         3
#define _T_Low          10.0            /** exec time **/
#define _T_Up           100.0

#define MaxResource     (_n_R_Up + 2)
#define MaxProcessor    (_n_P_Up + 2)

#endif  __GRCS_define_h_

:::::::::::::
solver/grcs.pgm/gen.c
:::::::::::::

void gen_sample_problem()
{ int i, j;

  /* generate the numbers of resources randomly */
  nTasks = problem.size;
  nProcessors = gen_random_int (_n_P_Low, _n_P_Up);
  nResources = gen_random_int (_n_R_Low, _n_R_Up);

  nProcessors = 2;
  nResources = 4;

  /* gen tasks' exec times randomly */
#ifdef SIMPLE_RCS
  for (i = 0; i < nTasks; i++) execTime[i] = 1.0;
#else
  for (i = 0; i < nTasks; i++) execTime[i] = gen_random_range (_T_Low, _T_Up);
#endif

  /** gen tasks' resources' requirements **/
  for (i = 0; i < nTasks; i++)
    for (j = 0; j < nResources; j++)
      REQ[i][j] = (gen_random_float () <= 0.5) ? 1 : 0;

  set_node_size ((nProcessors + nResources + nTasks) * sizeof (float) +
                 nTasks * sizeof (int));
}

:::::::::::::
solver/grcs.pgm/output.h
:::::::::::::
#ifndef __TS_output_h_
#define __TS_output_h_
```

```
/*
        Output Format Control Signals
        Every control signals must be defined to be either 1 or 0.
*/

#define         OUT_GRAPH               1
#define         OUT_SUMMARY             1

#define         OUT_TIME_LIMIT          1
#define         OUT_SPACE_LIMIT         0
#define         OUT_CST_LIMIT           0

#define         OUT_REAL_TIME           0
#define         OUT_REAL_MAX_SPACE      0
#define         OUT_REAL_CST            0

#define         OUT_VIRTUAL_TIME        1
#define         OUT_VIRTUAL_MAX_SPACE   1
#define         OUT_VIRTUAL_CST         1

#define         OUT_ROOT_APPROX         1
#define         OUT_RUN_TIME_APPROX     1
#define         OUT_APPROX              1

#define         OUT_INCUMBENT           1
#define         OUT_LOWB                1
#define         OUT_THRESHOLD           1

/*      sTCA & sTCGD & pTCGD            */
#define         OUT_GRADIENT_FACTOR     1

/*      pTCA & dTCA & pTCGD     */
#define         OUT_STOPPING_FACTOR     1
#define         OUT_CORRECTIVE_FACTOR   1

#endif __TS_output_h_

::::::::::::::
solver/grcs.pgm/search.c
::::::::::::::

typedef float decomp_;
decomp_ decomp_h_fun ();
decomp_ dh_val[MaxProblemSize];
node_ *ch_arr[MaxProblemSize];


node_ *expand (node, child_type, nchild)
  node_ *node;
  child_ child_type;
  int nchild;
{ int entity = node->entity;
  domain g_cost = node->g_cost;
  node_ *new_list = NULL;
  node_ *child, *next_child;
  int i, count = 0, who;
  domain w;
  decomp_ best;

  switch (child_type) {
  case ALL_CHILDREN:  nchild = HUGE_INT;
  case NEXT_N_CHILDREN:

    /** init **/
    for (i = 0; i < nTasks; i++) {
      dh_val[i] = (decomp_) 0;;
      ch_arr[i] = NULL;
    }

    /** sprout all necessary children **/
    for (i = 0; i < nTasks; i++)
      if (*(node->open_task+i)) {
        *(node->open_task+i) = 0;
        ch_arr[i] = allocate_node (node, i);
        dh_val[i] = decomp_h_fun (i);
        if (++count >= nchild) break;
      }

    /** sort all these children by decomp heuristic values **/
    while (count-- > 0) {
      best = (decomp_) 0;
      who = -1;
      for (i = 0; i < nTasks; i++)
        if (best < dh_val[i]) { best = dh_val[i]; who = i; }
      ch_arr[who]->next = new_list;
      new_list = ch_arr[who];
      dh_val[who] = (decomp_) 0;
    }

    /** reverse new_list **/
    next_child = new_list;
    new_list = NULL;
    while ((child = next_child)) {
      next_child = child->next;
      child->next = NULL;
      child->brother = new_list;
      new_list = child;
    }

    return new_list;
    break;

  case NEXT_CHILD:
    for (i = 0; i < nTasks; i++)
      if (*(node->open_task+i)) {
        *(node->open_task+i) = 0;
        return allocate_node (node, i);
      }
    break;

  default: break;
  }

  return NULL;
}


decomp_ decomp_h_fun (entity)
  int entity;
{
  return ((decomp_) execTime[entity]);
}


int is_infeasible (node)
  node_ *node;
{ return 0; }
```

```
int is_feasible (node)
  node_ *node;
{ int i;

  for (i = 0; i < nTasks; i++)
    if (is_task_alive (node, i)) return 0;

  return 1;
}

:::::::::::::
solver/grcs.pgm/support.c
:::::::::::::

node_ *root_generator () {
  return allocate_node(NULL, -1);
}

void iipd_init() { }

void idpd_init() { }

int is_dominated (node) node_ *node; { return 0; }


node_ *allocate_node (parentp, entity)
  node_ *parentp;
  int entity;
{ node_ *p;
  int i, j, who;
  domain tmax = (domain) 0.0;

  p = get_search_node_from_pool();
  init_node_struct(p, parentp);

  p->entity = entity;
  p->g_cost = (domain) 0.0;

  /* clean thses value fields */
  p->lowb = - problem.huge;
  p->upb = problem.huge;

  if (parentp)
    for (i = 0; i < nTasks; i++) *(p->sched_time+i) = *(parentp->sched_time+i);
  else
    for (i = 0; i < nTasks; i++) set_task_alive (p, i);

  for (i = 0; i < nTasks; i++)
    if (is_task_alive (p, i)) *(p->open_task+i) = 1;

  if (parentp) {
    for (i = 0; i < nProcessors; i++)
      *(p->proc_ck+i) = *(parentp->proc_ck+i);
  } else {
    for (i = 0; i < nProcessors; i++)
      *(p->proc_ck+i) = (domain) 0.0;
  }

  if (parentp) {
    for (i = 0; i < nResources; i++)
      *(p->resource_ck+i) = *(parentp->resource_ck+i);
  } else {
    for (i = 0; i < nResources; i++)
      *(p->resource_ck+i) = (domain) 0.0;
  }

  if (parentp) {
    tmax = alloc_proc (p->proc_ck, &who);

    for (j = 0; j < nResources; j++)
      if (REQ[entity][j])
        /** resource j is required **/
        if (tmax < *(p->resource_ck+j)) tmax = *(p->resource_ck+j);

    *(p->sched_time+entity) = tmax;
    tmax += execTime[entity];
    *(p->proc_ck+who) = tmax;

    for (j = 0; j < nResources; j++)
      if (REQ[entity][j]) *(p->resource_ck+j) = tmax;
  }

  tmax = (domain) 0.0;
  for (i = 0; i < nProcessors; i++)
    if (tmax < *(p->proc_ck+i)) tmax = *(p->proc_ck+i);

  p->g_cost = tmax;

  return p;
}


/* allocate associates for newly allocated search nodes */
void pd_region ()
{ int offset, *pool, pck_offset, rck_offset, open_offset, sched_time_offset;
  int *a, i;
  node_ *p;

  pool = (int *) pool_manager;
  offset = node_conf.node_size / sizeof(int);
  pck_offset = sizeof(node_) / sizeof(int);
  rck_offset = pck_offset + nProcessors * sizeof (domain) / sizeof (int);
  open_offset = rck_offset + nResources * sizeof (domain) / sizeof (int);
  sched_time_offset = open_offset + nTasks;

  for (i = 0; i < AllocationSize; i++) {
    p = ((node_ *) (a = pool + offset * i));
    p->proc_ck = (domain *) (a + pck_offset);
    p->resource_ck = (domain *) (a + rck_offset);
    p->open_task = (int *) (a + open_offset);
    p->sched_time = (domain *) (a + sched_time_offset);
  }
}


void reset_sol_buf (s)
  solution_ *s;
{ int i;

  for (i = 0; i < problem.size; i++) s->schedTime[i] = (domain) 0.0;
}


void evaluate_solution (sp)
  search_ *sp;
{ solution_ *sol;
  int i, j;
  if (sp) sol = sp->incumbent;
  else error("evaulate_solution: internal error: sp is nil");

  printf ("\nPROBLEM:\n");
  printf ("nTasks=%d, nProcessors=%d, nResources=%d\n",
          nTasks, nProcessors, nResources);

  for (i = 0; i < nTasks; i++)
    printf ("execTime[%d]=%g\n", i, (float) execTime[i]);

  for (i = 0; i < nTasks; i++) {
    for (j = 0; j < nResources; j++)
      printf ("REQ[T=%d][R=%d]=%d, ", i, j, REQ[i][j]);
    printf ("\n");
  }

  printf("\nSOLUTION:\n");
  for (i = 0; i < problem.size; i++)
    printf("%d-th task is scheduled at time  %g\n",
           i, (float) (sol->schedTime[i]));
}
```

```
Thu Jan 30 15:46:16 CST 1992
::::::::::::::
solver/maze.pgm/define.h
::::::::::::::

#ifndef __MAZE_define_h__
#define __MAZE_define_h__

/** this compiler definition is very essential, please don't delete it **/
#define DELAY_FREE_NODE

typedef long                    domain;
#define Domain                  LONG

#define AllocationSize          256
#define MaxProblemSize          130
#define PDSI_PART               int x, y;

#define MAX_X_SIZE              MaxProblemSize
#define MAX_Y_SIZE              MaxProblemSize

#if MaxProblemSize < 256
typedef char solelmt_;
#else
typedef int solelmt_;
#endif

typedef struct {
        int             pre_buf;
        solelmt_        x[MAX_X_SIZE * MAX_Y_SIZE];
        solelmt_        y[MAX_X_SIZE * MAX_Y_SIZE];
        solelmt_        dir[MAX_X_SIZE * MAX_Y_SIZE];
        int             post_buf;
} solution_;

#define EAST_WALL               (0x01)
#define SOUTH_WALL              (0x02)
#define WEST_WALL               (0x04)
#define NORTH_WALL              (0x08)
#define WALL                    (0x0f)

#define EAST_DOOR               (0x10)
#define SOUTH_DOOR              (0x20)
#define WEST_DOOR               (0x40)
#define NORTH_DOOR              (0x80)
#define DOOR                    (0xf0)


#define set_east_wall(pt)       pt |= EAST_WALL
#define set_south_wall(pt)      pt |= SOUTH_WALL
#define set_west_wall(pt)       pt |= WEST_WALL
#define set_north_wall(pt)      pt |= NORTH_WALL

#define set_east_door(pt)       pt |= EAST_DOOR
#define set_south_door(pt)      pt |= SOUTH_DOOR
#define set_west_door(pt)       pt |= WEST_DOOR
#define set_north_door(pt)      pt |= NORTH_DOOR


#define rm_east_wall(pt)        pt &= (~ EAST_WALL)
#define rm_south_wall(pt)       pt &= (~ SOUTH_WALL)
#define rm_west_wall(pt)        pt &= (~ WEST_WALL)
#define rm_north_wall(pt)       pt &= (~ NORTH_WALL)
```

```
#define rm_east_door(pt)        pt &= (~ EAST_DOOR)
#define rm_south_door(pt)       pt &= (~ SOUTH_DOOR)
#define rm_west_door(pt)        pt &= (~ WEST_DOOR)
#define rm_north_door(pt)       pt &= (~ NORTH_DOOR)


#define has_east_wall(pt)       (pt & EAST_WALL)
#define has_south_wall(pt)      (pt & SOUTH_WALL)
#define has_west_wall(pt)       (pt & WEST_WALL)
#define has_north_wall(pt)      (pt & NORTH_WALL)

#define has_east_door(pt)       (pt & EAST_DOOR)
#define has_south_door(pt)      (pt & SOUTH_DOOR)
#define has_west_door(pt)       (pt & WEST_DOOR)
#define has_north_door(pt)      (pt & NORTH_DOOR)

#define has_wall(pt)            (pt & WALL)
#define has_door(pt)            (pt & DOOR)


typedef enum {
        NOT_YET = 0,
        EAST = 1,
        SOUTH = 2,
        WEST = 3,
        NORTH = 4,
        NOT_USED_DIR = 100      /** for empty slot in solution **/
} directn;

typedef struct { int x, y; } coord;

#endif  __MAZE_define_h__

::::::::::::::
solver/maze.pgm/abc.c
::::::::::::::

int     X, Y;                   /** size of maze **/
int     X_start, Y_start;       /** start point **/
int     X_end, Y_end;           /** end point **/
int     x, y;                   /** current location of interest **/
int     idx_moves;              /** index of moves **/
#ifdef TAKEOUT_20
int     take_out_percent = 20;  /** percent of wall taking out **/
#else
int     take_out_percent = 35 /** 25 **/;       /** percent of wall taking out **/
#endif

directn Direction[] = { NOT_YET, EAST, SOUTH, WEST, NORTH };
directn AntiDirection[] = { NOT_YET, WEST, NORTH, EAST, SOUTH };

int     maze[MAX_X_SIZE][MAX_Y_SIZE];
coord   moves[MAX_X_SIZE * MAX_Y_SIZE];

::::::::::::::
solver/maze.pgm/bound.c
::::::::::::::

void evaluate_lower_bound (node)
    node_ *node;
{
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval lowb\n");
#endif
```

```
    node->lowb = node->g_cost + (domain) calc_goal_distance (node->x, node->y);
}


solution_ *evaluate_upper_bound (node, sol)
    node_ *node;
    solution_ *sol;
{   int count;
    node_ *p;
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval upb\n");
#endif

    if (is_feasible (node)) {
        /** calc how many moves so far **/
        for (count = 0, p = node; p; p = p->parent) count++;
        sol->dir[count] = (solelmt_) NOT_USED_DIR;

        for (p = node; p; p = p->parent) {
            --count;
            sol->x[count] = (solelmt_) p->x;
            sol->y[count] = (solelmt_) p->y;
            sol->dir[count] = (solelmt_) p->entity;
        }

        node->upb = node->g_cost;
    }
    else node->upb = maze_eval_upb (node, sol);

    return sol;
}


domain maze_eval_upb (node, sol)
    node_ *node;
    solution_ *sol;
{   directn curr_dir = node->entity;
    int curr_x = node->x;
    int curr_y = node->y;
    domain num_moves = 0;
    int i, who, new_x, new_y, count, new, best_x, best_y;
    directn new_dir, best_dir;
    domain best, dist;
    node_ *p;
    yesno_ pre_move ();

    /** calc how many moves so far **/
    for (count = 0, p = node; p; p = p->parent) count++;
    new = count;

    for (p = node; p; p = p->parent) {
        --count;
        sol->x[count] = (solelmt_) p->x;
        sol->y[count] = (solelmt_) p->y;
        sol->dir[count] = (solelmt_) p->entity;
    }

    num_moves = 0;
    moves[0].x = curr_x;
    moves[0].y = curr_y;
    while (1) {
        best = problem.huge;
        who = -1;
        for (i = 1; i <= 4; i++) {
            if (pre_move (curr_dir, curr_x, curr_y, i,
                          &new_dir, &new_x, &new_y, num_moves) == YES) {
                dist = calc_goal_distance (new_x, new_y);
                if (best > dist) {
                    best = dist;
                    who = i;
                    best_x = new_x;
                    best_y = new_y;
                    best_dir = new_dir;
                }
            }
        }

        if (who == -1) break;
        ++num_moves;
        if (num_moves + node->g_cost >= X * Y) { who = -1; break; }

        curr_dir = best_dir;
        moves[num_moves].x = curr_x = best_x;
        moves[num_moves].y = curr_y = best_y;

        sol->x[new] = (solelmt_) best_x;
        sol->y[new] = (solelmt_) best_y;
        sol->dir[new] = (solelmt_) best_dir;
        ++new;

        if (best == (domain) 0) break;
    }

    sol->dir[new] = (solelmt_) NOT_USED_DIR;

    return ((who == -1) ? problem.huge : (node->g_cost + num_moves));
}


yesno_ pre_move (curr_dir, curr_x, curr_y, which, new_dirp, new_xp, new_yp, num_moves)
    directn curr_dir;
    int curr_x, curr_y, which;
    directn *new_dirp;
    int *new_xp, *new_yp;
    domain num_moves;
{   yesno_ is_pre_movable ();

    if (curr_dir == AntiDirection[which]) return NO;

    if (is_pre_movable (curr_x, curr_y, Direction[which], num_moves) == YES) {
        switch ((*new_dirp = Direction[which])) {
        case EAST:  *new_xp = curr_x+1; *new_yp = curr_y;      break;
        case SOUTH: *new_xp = curr_x;   *new_yp = curr_y+1;    break;
        case WEST:  *new_xp = curr_x-1; *new_yp = curr_y;      break;
        case NORTH: *new_xp = curr_x;   *new_yp = curr_y-1;    break;
        }
        return YES;
    }

    return NO;
}


yesno_ is_pre_movable (x, y, dir, num_moves)
    int x, y;
    directn dir;
    domain num_moves;
```

```
{   int i;

    switch (dir) {
    case EAST:
        if (has_east_wall (maze[x][y])) return NO;
        if (++x >= X) return NO;
        break;
    case SOUTH:
        if (has_south_wall (maze[x][y])) return NO;
        if (++y >= Y) return NO;
        break;
    case WEST:
        if (has_west_wall (maze[x][y])) return NO;
        if (--x < 0) return NO;
        break;
    case NORTH:
        if (has_north_wall (maze[x][y])) return NO;
        if (--y < 0) return NO;
        break;
    default:
        return NO;
        break;
    }

    /** check whether this point has been visited or not **/
    for (i = 0; i <= num_moves; i++)
        if (x == moves[i].x && y == moves[i].y) return NO;

    return YES;
}


int calc_goal_distance (x, y)
    int x, y;
{   int d, error;

    d = x - X_end;
    if (d < 0) d = -d;
    error = d;

    d = y - Y_end;
    if (d < 0) d = -d;
    error += d;

    return error;
}

::::::::::::::
solver/maze.pgm/gen.c
::::::::::::::

int nrv_maze_conf;


void gen_sample_problem ()
{   int i, temp;
    directn door;
    FILE *fp, *fopen ();

    init_maze ();

    x = X_start;
    y = Y_start;
    idx_moves = 0;
```

```
    while (1) {
        moves[idx_moves].x = x;
        moves[idx_moves].y = y;

        /** select a new door **/
        while ((door = rand_door ()) == NOT_YET)
            /** no more door available, and backtrack **/
            if (backtrack () == NULL) {
#ifdef NRV_MAZE
            switch (nrv_maze_conf) {

            case 1: /** start at east wall **/
                set_east_wall (maze[X_start][Y_start]);
                set_west_wall (maze[X_end][Y_end]);
                break;

            case 2: /** start at south wall **/
                set_south_wall (maze[X_start][Y_start]);
                set_north_wall (maze[X_end][Y_end]);
                break;

            case 3: /** start at west wall **/
                set_west_wall (maze[X_start][Y_start]);
                set_east_wall (maze[X_end][Y_end]);
                break;

            case 4: /** start at north wall **/
                set_north_wall (maze[X_start][Y_start]);
                set_south_wall (maze[X_end][Y_end]);
                break;
            }

            /** new source and destination of NRV_MAZE **/
            X_start = X / 2;
            Y_start = Y / 2;
            X_end = gen_random_int (0, X-1);
            Y_end = gen_random_int (0, Y-1);
#else
#ifdef MAZE_ENHANCE
                enhance_maze ();
#endif
#ifdef BACKWARD_GEN
                /** swap  start point  and  end point **/
                temp = X_start; X_start = X_end; X_end = temp;
                temp = Y_start; Y_start = Y_end; Y_end = temp;
#endif
#endif

#ifdef DEBUG
if (cmd.debug >= 1)
    printf ("X_start=%d, Y_start=%d;  X_end=%d, Y_end=%d\n",
             X_start, Y_start, X_end, Y_end);
#endif

                /** PDSD size **/
                set_node_size (0);
                No_Upper_Bound = YES;

                fp = fopen (".maze", "a");
                fprintf (fp, "MAZE(%d,%d,%d)\n",
                                problem.size, problem.seed, take_out_percent);
                fclose (fp);
            return;
        }

        idx_moves++;

        /** rm the wall and set the door, and move it **/
        switch (door) {

        case EAST:
            rm_east_wall (maze[x][y]);
            set_east_door (maze[x][y]);
            x++;
            rm_west_wall (maze[x][y]);
            set_west_door (maze[x][y]);
            break;

        case SOUTH:
            rm_south_wall (maze[x][y]);
            set_south_door (maze[x][y]);
            y++;
            rm_north_wall (maze[x][y]);
            set_north_door (maze[x][y]);
            break;

        case WEST:
            rm_west_wall (maze[x][y]);
            set_west_door (maze[x][y]);
            x--;
            rm_east_wall (maze[x][y]);
            set_east_door (maze[x][y]);
            break;

        case NORTH:
            rm_north_wall (maze[x][y]);
            set_north_door (maze[x][y]);
            y--;
            rm_south_wall (maze[x][y]);
            set_south_door (maze[x][y]);
            break;

        default: error ("gen_sample_problem: no such direction"); break;
        }
    } /** end while **/
}


void init_maze ()
{   int i, j;

    X = Y = problem.size;

    /** init all maze points **/
    for (i = 0; i < X; i++)
        for (j = 0; j < Y; j++) maze[i][j] = 0;

    /** draw borders **/
    for (j = 0; j < Y; j++) set_east_wall (maze[X-1][j]);
    for (i = 0; i < X; i++) set_south_wall (maze[i][Y-1]);
    for (j = 0; j < Y; j++) set_west_wall (maze[0][j]);
    for (i = 0; i < X; i++) set_north_wall (maze[i][0]);

    /** randomly generate start and end points **/
    switch ((nrv_maze_conf = gen_random_int (1,4))) {
```

```
    case 1: /** start at east wall **/
        X_start = X-1;
        Y_start = gen_random_int (0, Y-1);
        X_end = 0;
        Y_end = gen_random_int (0, Y-1);
        rm_east_wall (maze[X_start][Y_start]);
        set_east_door (maze[X_start][Y_start]);
        rm_west_wall (maze[X_end][Y_end]);
        set_west_door (maze[X_end][Y_end]);
        break;

    case 2: /** start at south wall **/
        X_start = gen_random_int (0, X-1);
        Y_start = Y-1;
        X_end = gen_random_int (0, X-1);
        Y_end = 0;
        rm_south_wall (maze[X_start][Y_start]);
        set_south_door (maze[X_start][Y_start]);
        rm_north_wall (maze[X_end][Y_end]);
        set_north_door (maze[X_end][Y_end]);
        break;

    case 3: /** start at west wall **/
        X_start = 0;
        Y_start = gen_random_int (0, Y-1);
        X_end = X-1;
        Y_end = gen_random_int (0, Y-1);
        rm_west_wall (maze[X_start][Y_start]);
        set_west_door (maze[X_start][Y_start]);
        rm_east_wall (maze[X_end][Y_end]);
        set_east_door (maze[X_end][Y_end]);
        break;

    case 4: /** start at north wall **/
        X_start = gen_random_int (0, X-1);
        Y_start = 0;
        X_end = gen_random_int (0, X-1);
        Y_end = Y-1;
        rm_north_wall (maze[X_start][Y_start]);
        set_north_door (maze[X_start][Y_start]);
        rm_south_wall (maze[X_end][Y_end]);
        set_south_door (maze[X_end][Y_end]);
        break;
    }
}


directn rand_door () /** select a new door randomly **/
{   directn dir[4];
    int num_dir = 0;

    /** think about east **/
    if (! (has_east_door (maze[x][y]) || has_east_wall (maze[x][y]))) {
        if (has_door (maze[x+1][y]) && (x+1 != X_end || y != Y_end)) {
#ifdef MAZE_ENHANCE
            if (gen_random_int (1, 100) > take_out_percent) {
                set_east_wall (maze[x][y]);
                set_west_wall (maze[x+1][y]);
            }
#else
            set_east_wall (maze[x][y]);
            set_west_wall (maze[x+1][y]);
#endif
        } else dir[num_dir++] = EAST;
    }

    /** think about south **/
    if (! (has_south_door (maze[x][y]) || has_south_wall (maze[x][y]))) {
        if (has_door (maze[x][y+1]) && (x != X_end || y+1 != Y_end)) {
#ifdef MAZE_ENHANCE
            if (gen_random_int (1, 100) > take_out_percent) {
                set_south_wall (maze[x][y]);
                set_north_wall (maze[x][y+1]);
            }
#else
            set_south_wall (maze[x][y]);
            set_north_wall (maze[x][y+1]);
#endif
        } else dir[num_dir++] = SOUTH;
    }

    /** think about west **/
    if (! (has_west_door (maze[x][y]) || has_west_wall (maze[x][y]))) {
        if (has_door (maze[x-1][y]) && (x-1 != X_end || y != Y_end)) {
#ifdef MAZE_ENHANCE
            if (gen_random_int (1, 100) > take_out_percent) {
                set_west_wall (maze[x][y]);
                set_east_wall (maze[x-1][y]);
            }
#else
            set_west_wall (maze[x][y]);
            set_east_wall (maze[x-1][y]);
#endif
        } else dir[num_dir++] = WEST;
    }

    /** think about north **/
    if (! (has_north_door (maze[x][y]) || has_north_wall (maze[x][y]))) {
        if (has_door (maze[x][y-1]) && (x != X_end || y-1 != Y_end)) {
#ifdef MAZE_ENHANCE
            if (gen_random_int (1, 100) > take_out_percent) {
                set_north_wall (maze[x][y]);
                set_south_wall (maze[x][y-1]);
            }
#else
            set_north_wall (maze[x][y]);
            set_south_wall (maze[x][y-1]);
#endif
        } else dir[num_dir++] = NORTH;
    }

    switch (num_dir) {
    case 0: return NOT_YET; break;
    case 1: return dir[0]; break;
    case 2:
    case 3: return dir[gen_random_int (0, num_dir-1)]; break;
    default: fprintf (stderr, "rand_door: illegal num_dir = %d\n", num_dir);
             break;
    }
}


int backtrack ()
{
    idx_moves--;
    x = moves[idx_moves].x;
    y = moves[idx_moves].y;
    return idx_moves;
```

```
}

#ifdef MAZE_ENHANCE
void enhance_maze ()
/** make the maze much more difficult to solve **/
{
    /** make doors for all directions in start point **/

    if (X_start != X-1) {
        rm_east_wall (maze[X_start][Y_start]);
        rm_west_wall (maze[X_start+1][Y_start]);
    }

    if (Y_start != Y-1) {
        rm_south_wall (maze[X_start][Y_start]);
        rm_north_wall (maze[X_start][Y_start+1]);
    }

    if (X_start != 0) {
        rm_west_wall (maze[X_start][Y_start]);
        rm_east_wall (maze[X_start-1][Y_start]);
    }

    if (Y_start != 0) {
        rm_north_wall (maze[X_start][Y_start]);
        rm_south_wall (maze[X_start][Y_start-1]);
    }

    /** make doors for all directions in end point **/

    if (X_end != X-1) {
        rm_east_wall (maze[X_end][Y_end]);
        rm_west_wall (maze[X_end+1][Y_end]);
    }

    if (Y_end != Y-1) {
        rm_south_wall (maze[X_end][Y_end]);
        rm_north_wall (maze[X_end][Y_end+1]);
    }

    if (X_end != 0) {
        rm_west_wall (maze[X_end][Y_end]);
        rm_east_wall (maze[X_end-1][Y_end]);
    }

    if (Y_end != 0) {
        rm_north_wall (maze[X_end][Y_end]);
        rm_south_wall (maze[X_end][Y_end-1]);
    }
}
#endif

#define xform(loc)      (flag ? (loc) : -(loc))

void print_maze (fname, flag)
/** print maze layout in GRAP language **/
    char *fname;
    int flag;
{   int i, j;
    FILE *fp = fopen (fname, "w");

    fprintf (fp, ".G1\nframe invis ht 7i wid 7i\n");
    for (j = 0; j < Y; j++) {
        for (i = 0; i < X; i++) {
            if (has_east_wall (maze[i][j]))
                fprintf (fp, "line from %d, %d to %d, %d\n",
                        i+1, xform (j), i+1, xform (j+1));
            if (has_south_wall (maze[i][j]))
                fprintf (fp, "line from %d, %d to %d, %d\n",
                        i, xform (j+1), i+1, xform (j+1));
            if (has_west_wall (maze[i][j]))
                fprintf (fp, "line from %d, %d to %d, %d\n",
                        i, xform (j), i, xform (j+1));
            if (has_north_wall (maze[i][j]))
                fprintf (fp, "line from %d, %d to %d, %d\n",
                        i, xform (j), i+1, xform (j));
        }
    }

    if (flag) fprintf (fp, ".G2\n");
    fclose (fp);
}
::::::::::::::
solver/maze.pgm/search.c
::::::::::::::

node_ *expand (node, child_type, nchildren)
    node_ *node;
    child_ child_type;
    int nchildren;
{   node_ *head = NULL, *child;
    int i;

    /** check for infeasibility **/
    if (node->ndecomp >= 4) return NULL;

    switch (child_type) {
    case ALL_CHILDREN:
        for (i = 0; i < 4; i++)
            if (child = get_child (node, i)) {
                child->brother = head;
                head = child;
            }
        break;
    case NEXT_CHILD:
        if (head = get_child (node, node->ndecomp)) head->brother = NULL;
        break;
    case NEXT_N_CHILDREN:
        for (i = 0; i < nchildren; i++) {
            if (child = get_child (node, node->ndecomp)) {
                child->brother = head;
                head = child;
            }
            if (node->ndecomp >= 4) break;
        }
        break;
    default: break;
    }

    return head;
}


node_ *get_child (p, which)
    node_ *p;
    int which;
{   node_ *parentp = p->parent;
    int x = p->x, y = p->y;

    which++;  /** Direction[0] == AntiDirection[0] == NOT_YET **/
    (p->ndecomp)++;
    if (p->entity == AntiDirection[which]) return NULL;
    if (is_movable (p, x, y, Direction[which]) == YES)
        switch (Direction[which]) {
        case EAST:
            return (allocate_node (p, EAST, x+1, y, p->g_cost+1)); break;
        case SOUTH:
            return (allocate_node (p, SOUTH, x, y+1, p->g_cost+1)); break;
        case WEST:
            return (allocate_node (p, WEST, x-1, y, p->g_cost+1)); break;
        case NORTH:
            return (allocate_node (p, NORTH, x, y-1, p->g_cost+1)); break;
        default:
            error ("get_child: no such direction"); break;
        }

    return NULL;
}


yesno_ is_movable (p, x, y, dir)
    node_ *p;
    int x, y;
    directn dir;
{
    switch (dir) {
    case EAST:
        if (has_east_wall (maze[x][y])) return NO;
        if (++x >= X) return NO;
        break;
    case SOUTH:
        if (has_south_wall (maze[x][y])) return NO;
        if (++y >= Y) return NO;
        break;
    case WEST:
        if (has_west_wall (maze[x][y])) return NO;
        if (--x < 0) return NO;
        break;
    case NORTH:
        if (has_north_wall (maze[x][y])) return NO;
        if (--y < 0) return NO;
        break;
    default:    error ("is_movable: no such direction"); break;
    }

    /** check whether this point has been visited or not **/
    /** hence, we need DELAY_FREE_NODE compiler definition **/
    while ((p = p->parent))
        if (x == p->x && y == p->y) return NO;

    return YES;
}


int is_infeasible (node)
    node_ *node;
{ return 0; }


int is_feasible (node)
```

```
    node_ *node;
{ return ((node->x == X_end && node->y == Y_end) ? 1 : 0); }

::::::::::::::
solver/maze.pgm/support.c
::::::::::::::

node_ *root_generator ()
{ return allocate_node (NULL, NOT_YET, X_start, Y_start, (domain) 0); }

void iipd_init () {}
void idpd_init () { }

int is_dominated (node) node_ *node; { return 0; }


node_ *allocate_node (parentp, dir, x, y, g_cost)
    node_ *parentp;
    directn dir;
    int x, y;
    domain g_cost;
{   node_ *p;
    int i;

    p = get_search_node_from_pool ();
    init_node_struct (p, parentp);

    p->g_cost = g_cost;
    p->entity = dir;
    p->x = x;
    p->y = y;

    /** clean thses value fields **/
    p->lowb = p->upb = problem.huge;

    return p;
}


/** allocate associates for newly allocated search nodes **/
void pd_region () { }


void reset_sol_buf (s)
    solution_ *s;
{   int i, j;

    s->dir[0] = (solelmt_) NOT_USED_DIR;
}


void evaluate_solution (sp)
    search_ *sp;
{   solution_ *sol;
    node_ *p;
    directn dir;
    FILE *fp, *solfp;
    char fname[30];
    int count = 0, x, y;

    if (sp) sol = sp->incumbent;
    else error ("evaluate_solution: internal error: sp is nil");

    if (take_out_percent)
        sprintf (fname, "layout.%d.%d.%d",
                       problem.size, problem.seed, take_out_percent);
    else sprintf (fname, "layout.%d.%d", problem.size, problem.seed);

    print_maze (fname, 0);

    for (count = 0; ((directn) sol->dir[count]) != NOT_USED_DIR; count++) ;

    /** reverse the chain of moves **/
    fp = fopen (fname, "a");
    dir = NOT_YET;

    if (problem.size <= 30)

    while (--count >= 0) {
        x = sol->x[count];
        y = sol->y[count];
        switch (dir) {
        case EAST:
            fprintf (fp, "arrow from %f, %f to %f, %f\n",
                       x+.25, -(y+.5), x+.75, -(y+.5)); break;
        case SOUTH:
            fprintf (fp, "arrow from %f, %f to %f, %f\n",
                       x+.5, -(y+.25), x+.5, -(y+.75)); break;
        case WEST:
            fprintf (fp, "arrow from %f, %f to %f, %f\n",
                       x+.75, -(y+.5), x+.25, -(y+.5)); break;
        case NORTH:
            fprintf (fp, "arrow from %f, %f to %f, %f\n",
                       x+.5, -(y+.75), x+.5, -(y+.25)); break;
        case NOT_YET:
            fprintf (fp, "times at %f, %f\n", x+.5, -(y+.5)); break;
        }
        dir = (directn) sol->dir[count];
    }

    else

    while (--count >= 0)
        fprintf (fp, "dot at %f, %f\n", sol->x[count]+.5, -(sol->y[count]+.5));

    fprintf (fp, ".G2\n");
    fclose (fp);
}
```

```
Thu Jan 30 17:26:13 CST 1992
::::::::::::::
solver/puzz.pgm/define.h
::::::::::::::
#ifndef __PUZZ_define_h_
#define __PUZZ_define_h_

/** this compiler definition is essential, please don't delete it **/
#define DELAY_FREE_NODE

typedef long    domain;
#define Domain  LONG

#define AllocationSize          256

#define PDSI_PART          \
        int     *locmap;        /** location map of tiles **/

/** for problem generation **/
#define MaxProblemSize  30

#define NOT_USED_DIR    ('?')

/** solution is a pointer to node_ structure, however, it is not declared yet,
 ** as a result, "void *" is used instead. **/
typedef struct { int pre_buf; char *move; int post_buf; } solution_;

/** shorthand **/
#define get_x(loc)      (loc % LayoutSize1)
#define get_y(loc)      ((int) (loc / LayoutSize1))
#define get_loc(x,y)    (x + y * LayoutSize1)

#endif  __PUZZ_define_h_

::::::::::::::
solver/puzz.pgm/abc.c
::::::::::::::

int     *GoalLocMap;    /** goal location map of items **/
int     *InitLocMap;    /** goal location map of items **/
int     LayoutSize1;    /** size of board edge **/
int     LayoutSize2;    /** num of items, def= layout_size1 * layout_size1 **/

int     Direction[] = { 'N', 'S', 'E', 'W' };
int     AntiDirection[] = { 'S', 'N', 'W', 'E' };

int     *_tilemap;      /** internal var for printing board configuration **/
int     *tempLocMap;    /** temporary map of items **/

#ifdef KORF_BENCHMARKS
int     Korf[100][16] = {
{14,    13,     15,     7,      11,     12,     9,      5,      6,      0,      2,
1,      4,      8,      10,     3},
{13,    5,      4,      10,     9,      12,     8,      14,     2,      3,      7,
1,      0,      15,     11,     6},
{14,    7,      8,      2,      13,     11,     10,     4,      9,      12,     5,
0,      3,      6,      1,      15},
{5,     12,     10,     7,      15,     11,     14,     0,      8,      2,      1,
13,     3,      4,      9,      6},
{4,     7,      14,     13,     10,     3,      9,      12,     11,     5,      6,
15,     1,      2,      8,      0},
{14,    7,      1,      9,      12,     3,      6,      15,     8,      11,     2,
5,      10,     0,      4,      13},
{2,     11,     15,     5,      13,     4,      6,      7,      12,     8,      10,
```

```
1,      9,      3,      14,     0},
{12,    11,     15,     3,      8,      0,      4,      2,      6,      13,     9,      5
,       14,     1,      10,     7},
{3,     14,     9,      11,     5,      4,      8,      2,      13,     12,     6,      7
,       10,     1,      15,     0},
{13,    11,     8,      9,      0,      15,     7,      10,     4,      3,      6,      1
4,      5,      12,     2,      1},
{5,     9,      13,     14,     6,      3,      7,      12,     10,     8,      4,      0
,       15,     2,      11,     1},
{14,    1,      9,      6,      4,      8,      12,     5,      7,      2,      3,      0
,       10,     11,     13,     15},
{3,     6,      5,      2,      10,     0,      15,     14,     1,      4,      13,     1
2,      9,      8,      11,     7},
{7,     6,      8,      1,      11,     5,      14,     10,     3,      4,      9,      1
3,      15,     2,      0,      12},
{13,    11,     4,      12,     1,      8,      9,      15,     6,      5,      14,     2
,       7,      3,      10,     0},
{1,     3,      2,      5,      10,     9,      15,     6,      8,      14,     13,     1
1,      12,     4,      7,      0},
{15,    14,     0,      4,      11,     1,      6,      13,     7,      5,      8,      9
,       3,      2,      10,     12},
{6,     0,      14,     12,     1,      15,     9,      10,     11,     4,      7,      2
,       8,      3,      5,      13},
{7,     11,     8,      3,      14,     0,      6,      15,     1,      4,      13,     9
,       5,      12,     2,      10},
{6,     12,     11,     3,      13,     7,      9,      15,     2,      14,     8,      1
0,      4,      1,      5,      0},
{12,    8,      14,     6,      11,     4,      7,      0,      5,      1,      10,     1
5,      3,      13,     9,      2},
{14,    3,      9,      1,      15,     8,      4,      5,      11,     7,      10,     1
3,      0,      2,      12,     6},
{10,    9,      3,      11,     0,      13,     2,      14,     5,      6,      4,      7
,       8,      15,     1,      12},
{7,     3,      14,     13,     4,      1,      10,     8,      5,      12,     9,      1
1,      2,      15,     6,      0},
{11,    4,      2,      7,      1,      0,      10,     15,     6,      9,      14,     8
,       3,      13,     5,      12},
{5,     7,      3,      12,     15,     13,     14,     8,      0,      10,     9,      6
,       1,      4,      2,      11},
{14,    1,      8,      15,     2,      6,      0,      3,      9,      12,     10,     1
3,      4,      7,      5,      11},
{13,    14,     6,      12,     4,      5,      1,      0,      9,      3,      10,     2
,       15,     11,     8,      7},
{9,     8,      0,      2,      15,     1,      4,      14,     3,      10,     7,      5
,       11,     13,     6,      12},
{12,    15,     2,      6,      1,      14,     4,      8,      5,      3,      7,      0
,       10,     13,     9,      11},
{12,    8,      15,     13,     1,      0,      5,      4,      6,      3,      2,      1
1,      9,      7,      14,     10},
{14,    10,     9,      4,      13,     6,      5,      8,      2,      12,     7,      0
,       1,      3,      11,     15},
{14,    3,      5,      15,     11,     6,      13,     9,      0,      10,     2,      1
2,      4,      1,      7,      8},
{6,     11,     7,      8,      13,     2,      5,      4,      1,      10,     3,      9
,       14,     0,      12,     15},
{1,     6,      12,     14,     3,      2,      15,     8,      4,      5,      13,     9
,       0,      7,      11,     10},
{12,    6,      0,      4,      7,      3,      15,     1,      13,     9,      8,      1
1,      2,      14,     5,      10},
{8,     1,      7,      12,     11,     0,      10,     5,      9,      15,     6,      1
3,      14,     2,      3,      4},
{7,     15,     8,      2,      13,     6,      3,      12,     11,     0,      4,      1
0,      9,      5,      1,      14},
{9,     0,      4,      10,     1,      14,     15,     3,      12,     6,      5,      7
```

```
,       11,     13,     8,      2},
{11,    5,      1,      14,     4,      12,     10,     0,      2,      7,      13,
3,      9,      15,     6,      8},
{8,     13,     10,     9,      11,     3,      15,     6,      0,      1,      2,
14,     12,     5,      4,      7},
{4,     5,      7,      2,      9,      14,     12,     13,     0,      3,      6,
11,     8,      1,      15,     10},
{11,    15,     14,     13,     1,      9,      10,     4,      3,      6,      2,
12,     7,      5,      8,      0},
{12,    9,      0,      6,      8,      3,      5,      14,     2,      4,      11,
7,      10,     1,      15,     13},
{3,     14,     9,      7,      12,     15,     0,      4,      1,      8,      5,
6,      11,     10,     2,      13},
{8,     4,      6,      1,      14,     12,     2,      15,     13,     10,     9,
5,      3,      7,      0,      11},
{6,     10,     1,      14,     15,     8,      3,      5,      13,     0,      2,
7,      4,      9,      11,     12},
{8,     11,     4,      6,      7,      3,      10,     9,      2,      12,     15,
13,     0,      1,      5,      14},
{10,    0,      2,      4,      5,      1,      6,      12,     11,     13,     9,
7,      15,     3,      14,     8},
{12,    5,      13,     11,     2,      10,     0,      9,      7,      8,      4,
3,      14,     6,      15,     1},
{10,    2,      8,      4,      15,     0,      1,      14,     11,     13,     3,
6,      9,      7,      5,      12},
{10,    8,      0,      12,     3,      7,      6,      2,      1,      14,     4,
11,     15,     13,     9,      5},
{14,    9,      12,     13,     15,     4,      8,      10,     0,      2,      1,
7,      3,      11,     5,      6},
{12,    11,     0,      8,      10,     2,      13,     15,     5,      4,      7,
3,      6,      9,      14,     1},
{13,    8,      14,     3,      9,      1,      0,      7,      15,     5,      4,
10,     12,     2,      6,      11},
{3,     15,     2,      5,      11,     6,      4,      7,      12,     9,      1,
0,      13,     14,     10,     8},
{5,     11,     6,      9,      4,      13,     12,     0,      8,      2,      15,
10,     1,      7,      3,      14},
{5,     0,      15,     8,      4,      6,      1,      14,     10,     11,     3,
9,      7,      12,     2,      13},
{15,    14,     6,      7,      10,     1,      0,      11,     12,     8,      4,
9,      2,      5,      13,     3},
{11,    14,     13,     1,      2,      3,      12,     4,      15,     7,      9,
5,      10,     6,      8,      0},
{6,     13,     3,      2,      11,     9,      5,      10,     1,      7,      12,
14,     8,      4,      0,      15},
{4,     6,      12,     0,      14,     2,      9,      13,     11,     8,      3,
15,     7,      10,     1,      5},
{8,     10,     9,      11,     14,     1,      7,      15,     13,     4,      0,
12,     6,      2,      5,      3},
{5,     2,      14,     0,      7,      8,      6,      3,      11,     12,     13,
15,     4,      10,     9,      1},
{7,     8,      3,      2,      10,     12,     4,      6,      11,     13,     5,
15,     0,      1,      9,      14},
{11,    6,      14,     12,     3,      5,      1,      15,     8,      0,      10,
13,     9,      7,      4,      2},
{7,     1,      2,      4,      8,      3,      6,      11,     10,     15,     0,
5,      14,     12,     13,     9},
{7,     3,      1,      13,     12,     10,     5,      2,      8,      0,      6,
11,     14,     15,     4,      9},
{6,     0,      5,      15,     1,      14,     4,      9,      2,      13,     8,
10,     11,     12,     7,      3},
{15,    1,      3,      12,     4,      0,      6,      5,      2,      8,      14,
9,      13,     10,     7,      11},
{5,     7,      0,      11,     12,     1,      9,      10,     15,     6,      2,
```

```
3,      8,      4,      13,     14},
{12,    15,     11,     10,     4,      5,      14,     0,      13,     7,      1,      2
,       9,      8,      3,      6},
{6,     14,     10,     5,      15,     8,      7,      1,      3,      4,      2,      0
,       12,     9,      11,     13},
{14,    13,     4,      11,     15,     8,      6,      9,      0,      7,      3,      1
,       2,      10,     12,     5},
{14,    4,      0,      10,     6,      5,      1,      3,      9,      2,      13,     1
5,      12,     7,      8,      11},
{15,    10,     8,      3,      0,      6,      9,      5,      1,      14,     13,     1
1,      7,      2,      12,     4},
{0,     13,     2,      4,      12,     14,     6,      9,      15,     1,      10,     3
,       11,     5,      8,      7},
{3,     14,     13,     6,      4,      15,     8,      9,      5,      12,     10,     0
,       2,      7,      1,      11},
{0,     1,      9,      7,      11,     13,     5,      3,      14,     12,     4,      2
,       8,      6,      10,     15},
{11,    0,      15,     8,      13,     12,     3,      5,      10,     1,      4,      6
,       14,     9,      7,      2},
{13,    0,      9,      12,     11,     6,      3,      5,      15,     8,      1,      1
0,      4,      14,     2,      7},
{14,    10,     2,      1,      13,     9,      8,      11,     7,      3,      6,      1
2,      15,     5,      4,      0},
{12,    3,      9,      1,      4,      5,      10,     2,      6,      11,     15,     0
,       14,     7,      13,     8},
{15,    8,      10,     7,      0,      12,     14,     1,      5,      9,      6,      3
,       13,     11,     4,      2},
{4,     7,      13,     10,     1,      2,      9,      6,      12,     8,      14,     5
,       3,      0,      11,     15},
{6,     0,      5,      10,     11,     12,     9,      2,      1,      7,      4,      3
,       14,     8,      13,     15},
{9,     5,      11,     10,     13,     0,      2,      1,      8,      6,      14,     1
2,      4,      7,      3,      15},
{15,    2,      12,     11,     14,     13,     9,      5,      1,      3,      8,      7
,       0,      10,     6,      4},
{11,    1,      7,      4,      10,     13,     3,      8,      9,      14,     0,      1
5,      6,      5,      2,      12},
{5,     4,      7,      1,      11,     12,     14,     15,     10,     13,     8,      6
,       2,      0,      9,      3},
{9,     7,      5,      2,      14,     15,     12,     10,     11,     3,      6,      1
,       8,      13,     0,      4},
{3,     2,      7,      9,      0,      15,     12,     4,      6,      11,     5,      1
4,      8,      13,     10,     1},
{13,    9,      14,     6,      12,     8,      1,      2,      3,      4,      0,      7
,       5,      10,     11,     15},
{5,     7,      11,     8,      0,      14,     9,      13,     10,     12,     3,      1
5,      6,      1,      4,      2},
{4,     3,      6,      13,     7,      15,     9,      0,      10,     5,      8,      1
1,      2,      12,     1,      14},
{1,     7,      15,     14,     2,      6,      4,      9,      12,     11,     13,     3
,       0,      8,      5,      10},
{9,     14,     5,      7,      8,      15,     1,      2,      10,     4,      13,     6
,       12,     0,      11,     3},
{0,     11,     3,      12,     5,      2,      1,      9,      8,      10,     14,     1
5,      7,      4,      13,     6},
{7,     15,     4,      0,      10,     9,      2,      5,      12,     11,     13,     6
,       1,      3,      14,     8},
{11,    4,      0,      8,      6,      10,     5,      13,     12,     7,      14,     3
,       1,      2,      9,      15}
};
#endif
::::::::::::::
solver/puzz.pgm/bound.c
::::::::::::::
```

```
domain puzz_eval_lowb (), puzz_eval_upb ();


void evaluate_lower_bound (node)
    node_ *node;
{
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval lowb\n");
#endif

    node->lowb = node->g_cost + (domain) calc_distance (node->locmap);
}


solution_ *evaluate_upper_bound (node, sol)
    node_ *node;
    solution_ *sol;
{   node_ *p;
    int count;
#ifdef DEBUG
if (cmd.debug >= 4) printf("eval upb\n");
#endif

    sol->move = NULL;
    if (is_feasible (node)) {
        node->upb = node->g_cost;

        for (count = 0, p = node; p->entity != -1; p = p->parent) count++;
        sol->move = (char *) malloc ((count+1) * sizeof (char));
        *(sol->move+count) = NOT_USED_DIR;
        for (--count, p = node; p->entity != -1; p = p->parent, --count)
            *(sol->move+count) = (char) p->entity;
    }
    else node->upb = problem.huge;

    return sol;
}


int calc_distance (locmap)
    int *locmap;
{   int i, d, error = 0, x1, x2, y1, y2;

    for (i = 1; i < LayoutSize2; i++) {
        x1 = get_x (*(GoalLocMap + i));
        y1 = get_y (*(GoalLocMap + i));
        x2 = get_x (*(locmap + i));
        y2 = get_y (*(locmap + i));
        if ((d = x1 - x2) < 0) d = -d;
        error += d;
        if ((d = y1 - y2) < 0) d = -d;
        error += d;
    }
    return error;
}

::::::::::::::
solver/puzz.pgm/gen.c
::::::::::::::

void gen_sample_problem ()
{   int i, j, temp, who, bound;

#ifdef KORF_BENCHMARKS
    static int index = -1000;
    if (index == -1000) index = problem.seed - 101;
#endif

    LayoutSize1 = problem.size;
    LayoutSize2 = LayoutSize1 * LayoutSize1;
    bound = LayoutSize2 - 1;

#if defined (TEST_KORF_BENCHMARKS) && defined (KORF_BENCHMARKS)
    /** test if any typo in Korf's 15-puzzle benchmarks **/
    for (index = 0; index < 100; index++)
        for (i = 0; i < LayoutSize2; i++) {
            if (Korf[index][i] > 15 && Korf[index][i] < 0)
                printf ("gen: Korf's 15-puzzle benchmark: out of range: index=%d, i=%d\n"
, index, i);
            for (j = i+1; j < LayoutSize2; j++)
                if (Korf[index][i] == Korf[index][j])
                    printf ("gen: Korf's 15-puzzle benchmark: duplicate: index=%d, i=%d,
j=%d\n", index, i, j);
        }
    printf ("Congratulations: PASSED\n");
    exit (0);
#endif

    /** gen initial loc map of items **/
    InitLocMap = (int *) malloc (sizeof (int) * LayoutSize2);

    if (problem.seed <= 100 || problem.seed > 200 || problem.size != 4) {
        /** my own way to generate problem **/
        for (i = 0; i < LayoutSize2; i++) *(InitLocMap + i) = i;
        for (i = 0; i < bound; i++) {
            who = gen_random_int (i, bound);
            temp = *(InitLocMap + i);
            *(InitLocMap + i) = *(InitLocMap + who);
            *(InitLocMap + who) = temp;
        }
    } else {
        /** use Korf's 15-puzzle benchmarks in IDA* paper for seed in [101,200] **/
        for (i = 0; i < LayoutSize2; i++) *(InitLocMap + Korf[index][i]) = i;
        index++;
    }

    /** gen goal loc map of items **/
    GoalLocMap = (int *) malloc (sizeof (int) * LayoutSize2);
    for (i = 0; i < LayoutSize2; i++) *(GoalLocMap + i) = i;

    /** gen temp loc map **/
    tempLocMap = (int *) malloc (sizeof (int) * LayoutSize2);

#ifdef DEBUG
if (cmd.debug >= 1) {
    printf ("Initial State:\n");
    for (i = 0; i < LayoutSize2; i++)
        *(tempLocMap + *(InitLocMap + i)) = i;
    for (i = 0; i < LayoutSize2; i++)
        printf (" %d ", *(tempLocMap + i));
    printf ("\nGoal State:\n");
    for (i = 0; i < LayoutSize2; i++)
        *(tempLocMap + *(GoalLocMap + i)) = i;
    for (i = 0; i < LayoutSize2; i++)
        printf (" %d ", *(tempLocMap + i));
    printf ("\n");
}
#endif
```

```
    /** PDSD size **/
    set_node_size (LayoutSize2 * sizeof (int));

    /** PUZZ does not supply upper bound **/
    No_Upper_Bound = YES;
}

::::::::::::::
solver/puzz.pgm/search.c
::::::::::::::

node_ *get_child ();
yesno_ is_movable (), is_same_layout ();
void move_blank (), move_locmap_blank ();


node_ *expand (node, child_type, nchildren)
    node_ *node;
    child_ child_type;
    int nchildren;
{   node_ *head = NULL, *child;
    int i;

    /** check for infeasibility **/
    if (node->ndecomp >= 4) return NULL;

    switch (child_type) {
    case ALL_CHILDREN:
        for (i = 0; i < 4; i++)
            if (child = get_child (node, i)) {
                child->brother = head;
                head = child;
            }
        break;
    case NEXT_CHILD:
        if (head = get_child (node, node->ndecomp)) head->brother = NULL;
        break;
    case NEXT_N_CHILDREN:
        for (i = 0; i < nchildren; i++) {
            if (child = get_child (node, node->ndecomp)) {
                child->brother = head;
                head = child;
            }
            if (node->ndecomp >= 4) break;
        }
        break;
    default: break;
    }

    return head;
}


node_ *get_child (node, which)
    node_ *node;
    int which;
{   node_ *parentp = node->parent;

    (node->ndecomp)++;
    if (node->entity == AntiDirection[which]) return NULL;
    if (is_movable (node, Direction[which]) == YES)
        return allocate_node (node, Direction[which], node->g_cost+1);
    return NULL;
}


yesno_ is_movable (node, dir)
    node_ *node;
    int dir;
{   int x = get_x (*(node->locmap));
    int y = get_y (*(node->locmap));
    int i;
    node_ *p;

    switch (dir) {
    case 'N': if (++y >= LayoutSize1)   return NO; break;
    case 'S': if (--y < 0)              return NO; break;
    case 'E': if (++x >= LayoutSize1)   return NO; break;
    case 'W': if (--x < 0)              return NO; break;
    default:  error ("is_movable: no such direction"); break;
    }

    /** check the history **/
    for (i = 0; i < LayoutSize2; i++) *(tempLocMap+i) = *(node->locmap+i);
    move_locmap_blank (tempLocMap, dir);
    for (p = node; p; p = p->parent)
        if (is_same_layout (tempLocMap, p->locmap)) return NO;

    return YES;
}


void move_blank (node, dir)
    node_ *node;
    int dir;
{
    move_locmap_blank (node->locmap, dir);
}


void move_locmap_blank (locmap, dir)
    int locmap[];
    int dir;
{   int x = get_x (locmap[0]);
    int y = get_y (locmap[0]);
    int i, where, temp, err_flag = 0;

    switch (dir) {
    case 'N': if (++y >= LayoutSize1)   err_flag = 1; break;
    case 'S': if (--y < 0)              err_flag = 1; break;
    case 'E': if (++x >= LayoutSize1)   err_flag = 1; break;
    case 'W': if (--x < 0)              err_flag = 1; break;
    default:                            err_flag = 1; break;
    }
    if (err_flag) error ("is_movable: no such direction");

    where = get_loc (x, y);
    for (i = 0; i < LayoutSize2; i++)
        if (where == locmap[i]) break;
    if (i == LayoutSize2) error ("move_blank: cannot move");
    temp = locmap[0];
    locmap[0] = locmap[i];
    locmap[i] = temp;
}


int is_infeasible (node)
```

```
    node_ *node;
{ return 0; }


int is_feasible (node)
    node_ *node;
{ return (is_same_layout (node->locmap, GoalLocMap)); }


yesno_ is_same_layout (locmap1, locmap2)
    int *locmap1, *locmap2;
{   int i;

    for (i = 0; i < LayoutSize2; i++)
        if (*(locmap1 + i) != *(locmap2 + i)) return NO;
    return YES;
}

:::::::::::::
solver/puzz.pgm/support.c
:::::::::::::

node_ *root_generator () { return (allocate_node (NULL, -1, (domain) 0)); }

void iipd_init ()
{ _tilemap = (int *) malloc (problem.size * problem.size * sizeof (int)); }

void idpd_init () { }

int is_dominated (node) node_ *node; { return 0; }


node_ *allocate_node (parentp, entity, g_cost)
    node_ *parentp;
    int entity;
    domain g_cost;
{   node_ *p;
    int i;

    p = get_search_node_from_pool();
    init_node_struct(p, parentp);

    p->entity = entity;
    p->g_cost = g_cost;

    /** clean thses value fields **/
    p->lowb = - problem.huge;
    p->upb = problem.huge;

    if (parentp) {
        for (i = 0; i < LayoutSize2; i++)
            *(p->locmap + i) = *(parentp->locmap + i);
        move_blank (p, entity);
    } else {
        for (i = 0; i < LayoutSize2; i++)
            *(p->locmap + i) = *(InitLocMap + i);
    }

    return p;
}


/** allocate associates for newly allocated search nodes **/
void pd_region ()
{   int offset, *pool, locmap_offset, *a, i;
    node_ *p;

    pool = (int *) pool_manager;
    offset = node_conf.node_size / sizeof (int);
    locmap_offset = sizeof (node_) / sizeof (int);
    for (i = 0; i < AllocationSize; i++) {
        p = ((node_ *) (a = pool + offset * i));
        p->locmap = a + locmap_offset;  /** allocate *locmap frame **/
    }
}


void reset_sol_buf (s) solution_ *s; { s->move = NULL; }


void print_conf (locmap)
    int locmap[];
{   int x, y, loc, tile;

    /** transform locmap into tilemap **/
    for (tile = 0; tile < LayoutSize2; tile++)
        *(_tilemap + locmap[tile]) = tile;

    /** print out the board based on tilemap **/
    printf ("\n");
    for (y = LayoutSize1 - 1; y >= 0; y--) {
        for (x = 0; x < LayoutSize1; x++) {
            loc = get_loc (x, y);
            printf (" %2d  ", *(_tilemap + loc));
        }
        printf ("\n");
    }
}


void evaluate_solution (sp)
    search_ *sp;
{   solution_ *sol;
    int i;

    if (sp) sol = sp->incumbent;
    else error ("evaluate_solution: internal error: sp is nil");

    printf ("\nInitial Board");
    print_conf (InitLocMap);
    printf ("Goal Board");
    print_conf (GoalLocMap);
    printf ("\nSOLUTION:\n");

    for (i = 0; *(sol->move+i) != NOT_USED_DIR; i++) {
#ifdef FULL_DISPLAY
        printf ("\n%c", (char) *(sol->move+i));
        move_locmap_blank (InitLocMap, (int) *(sol->move+i));
        print_conf (InitLocMap);
#else
        printf (" %c ", (char) *(sol->move+i));
#endif
    }
    printf ("\n");
}
```

ISE'ts'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:38 CST 1992
::::::::::::::
ts.Makefile
::::::::::::::
.SILENT:
CC = cc
CFlags = -DDEBUG -DSUN -DSYM_TSP -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = ts
PROBM = $(SRC)/solver/ats.pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
        $(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
        cat $(HelpFiles) $(ALG)/*.c > _;\
        mv -f _ .algorithm.c;\
        $(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
        cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
        mv -f _ .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
        cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
        mv -f _ .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
        cat $(HelpFiles) $(KERNEL)/*.c > _;\
        mv -f _ .kernel.c;\
        $(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
        cat $(HelpFiles) $(OPEN)/*.c > _;\
        mv -f _ .open.c;\
        $(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
        cat $(HelpFiles) $(PRIM)/*.c > _;\
        mv -f _ .primitive.c;\
        $(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
        cat $(HelpFiles) $(PROBM)/*.c > _;\
        mv -f _ .problem.c;\
        $(CC) $(CFlags) -c .problem.c -o problem.o
```

ISE'ks'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:35 CST 1992
::::::::::::::
ks.Makefile
::::::::::::::
.SILENT:
CC = cc
CFlags = -DDEBUG -DSUN -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = ks
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
	$(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
	rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
	cat $(HelpFiles) $(ALG)/*.c > _;\
	mv -f _ .algorithm.c;\
	$(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
	cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
	mv -f _ .include.c;\
	$(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
	cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
	mv -f _ .interface.c;\
	$(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
	cat $(HelpFiles) $(KERNEL)/*.c > _;\
	mv -f _ .kernel.c;\
	$(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
	cat $(HelpFiles) $(OPEN)/*.c > _;\
	mv -f _ .open.c;\
	$(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
	cat $(HelpFiles) $(PRIM)/*.c > _;\
	mv -f _ .primitive.c;\
	$(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
	cat $(HelpFiles) $(PROBM)/*.c > _;\
	mv -f _ .problem.c;\
	$(CC) $(CFlags) -c .problem.c -o problem.o
```

ISE'pp'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:32 CST 1992
::::::::::::::
pp.Makefile
::::::::::::::
.SILENT:
CC = cc
CFlags = -DDEBUG -DSUN -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = pp
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
        $(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
        cat $(HelpFiles) $(ALG)/*.c > _;\
        mv -f _ .algorithm.c;\
        $(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
        cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
        mv -f _ .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
        cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
        mv -f _ .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
        cat $(HelpFiles) $(KERNEL)/*.c > _;\
        mv -f _ .kernel.c;\
        $(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
        cat $(HelpFiles) $(OPEN)/*.c > _;\
        mv -f _ .open.c;\
        $(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
        cat $(HelpFiles) $(PRIM)/*.c > _;\
        mv -f _ .primitive.c;\
        $(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
        cat $(HelpFiles) $(PROBM)/*.c > _;\
        mv -f _ .problem.c;\
        $(CC) $(CFlags) -c .problem.c -o problem.o
```

ISE'vc'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:29 CST 1992
::::::::::::::
vc.Makefile
::::::::::::::
.SILENT:
CC = cc
CFlags = -DDEBUG -DSUN -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = vc
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
        $(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
        cat $(HelpFiles) $(ALG)/*.c > _;\
        mv -f _ .algorithm.c;\
        $(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
        cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
        mv -f _ .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
        cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
        mv -f _ .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
        cat $(HelpFiles) $(KERNEL)/*.c > _;\
        mv -f _ .kernel.c;\
        $(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
        cat $(HelpFiles) $(OPEN)/*.c > _;\
        mv -f _ .open.c;\
        $(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
        cat $(HelpFiles) $(PRIM)/*.c > _;\
        mv -f _ .primitive.c;\
        $(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
        cat $(HelpFiles) $(PROBM)/*.c > _;\
        mv -f _ .problem.c;\
        $(CC) $(CFlags) -c .problem.c -o problem.o
```

ISE'wct'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:27 CST 1992
::::::::::::::
wct.Makefile
::::::::::::::
.SILENT:
CC = cc
CFlags = -DDEBUG -DSUN -DLOWB_GUIDANCE -g
LoadFlags = -lm -lg

SRC = ../../src

CSP = wct
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
        $(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
        cat $(HelpFiles) $(ALG)/*.c > _;\
        mv -f _ .algorithm.c;\
        $(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
        cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
        mv -f _ .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
        cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
        mv -f _ .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
        cat $(HelpFiles) $(KERNEL)/*.c > _;\
        mv -f _ .kernel.c;\
        $(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
        cat $(HelpFiles) $(OPEN)/*.c > _;\
        mv -f _ .open.c;\
        $(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
        cat $(HelpFiles) $(PRIM)/*.c > _;\
        mv -f _ .primitive.c;\
        $(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
        cat $(HelpFiles) $(PROBM)/*.c > _;\
        mv -f _ .problem.c;\
        $(CC) $(CFlags) -c .problem.c -o problem.o
```

ISE'grcs'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:24 CST 1992
::::::::::::::
grcs.Makefile
::::::::::::::
.SILENT:
CC = cc
CFlags = -DDEBUG -DSUN -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = grcs
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
	$(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
	rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
	cat $(HelpFiles) $(ALG)/*.c > _;\
	mv -f _ .algorithm.c;\
	$(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
	cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
	mv -f _ .include.c;\
	$(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
	cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
	mv -f _ .interface.c;\
	$(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
	cat $(HelpFiles) $(KERNEL)/*.c > _;\
	mv -f _ .kernel.c;\
	$(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
	cat $(HelpFiles) $(OPEN)/*.c > _;\
	mv -f _ .open.c;\
	$(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
	cat $(HelpFiles) $(PRIM)/*.c > _;\
	mv -f _ .primitive.c;\
	$(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
	cat $(HelpFiles) $(PROBM)/*.c > _;\
	mv -f _ .problem.c;\
	$(CC) $(CFlags) -c .problem.c -o problem.o
```

ISE'ats'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:18 CST 1992
::::::::::::::
ats.Makefile
::::::::::::::
.SILENT:
CC = cc
CFlags = -DDEBUG -DSUN -DASYM_TSP -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = ats
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
        $(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
        cat $(HelpFiles) $(ALG)/*.c > _;\
        mv -f _ .algorithm.c;\
        $(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
        cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
        mv -f _ .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
        cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
        mv -f _ .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
        cat $(HelpFiles) $(KERNEL)/*.c > _;\
        mv -f _ .kernel.c;\
        $(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
        cat $(HelpFiles) $(OPEN)/*.c > _;\
        mv -f _ .open.c;\
        $(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
        cat $(HelpFiles) $(PRIM)/*.c > _;\
        mv -f _ .primitive.c;\
        $(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
        cat $(HelpFiles) $(PROBM)/*.c > _;\
        mv -f _ .problem.c;\
        $(CC) $(CFlags) -c .problem.c -o problem.o
```

```
Thu Jan 30 17:15:05 CST 1992
::::::::::::::
maze.Makefile
::::::::::::::
.SILENT:
# BACKWARD_GEN
# MAZE_ENHANCE
CC = cc
CFlags = -DDEBUG -DSUN -DTAKEOUT_20 -DBACKWARD_GEN -DMAZE_ENHANCE -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = maze
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
        $(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
        cat $(HelpFiles) $(ALG)/*.c > _;\
        mv -f _ .algorithm.c;\
        $(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
        cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
        mv -f _ .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
        cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
        mv -f _ .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
        cat $(HelpFiles) $(KERNEL)/*.c > _;\
        mv -f _ .kernel.c;\
        $(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
        cat $(HelpFiles) $(OPEN)/*.c > _;\
        mv -f _ .open.c;\
        $(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
        cat $(HelpFiles) $(PRIM)/*.c > _;\
        mv -f _ .primitive.c;\
        $(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
        cat $(HelpFiles) $(PROBM)/*.c > _;\
        mv -f _ .problem.c;\
        $(CC) $(CFlags) -c .problem.c -o problem.o
```

ISE'puzz'Makefile, CRHC-92-1

```
Thu Jan 30 17:14:14 CST 1992
::::::::::::::
puzz.Makefile
::::::::::::::
.SILENT:
# FULL_DISPLAY
CC = cc
CFlags = -DDEBUG -DSUN -DFULL_DISPLAY -DLOWB_GUIDANCE -g
LoadFlags = -lm

SRC = ../../src

CSP = puzz
PROBM = $(SRC)/solver/$(CSP).pgm
BIN = .

HOME = $(SRC)
ALG = $(HOME)/algorithm
INCL = $(HOME)/include
INTFC = $(HOME)/interface
KERNEL = $(HOME)/kernel
OPEN = $(HOME)/open
PRIM = $(HOME)/primitive

ConfigFiles = $(PROBM)/config.h $(INCL)/config.h

MostHelpFiles = $(INCL)/limits.h $(PROBM)/define.h $(INCL)/define.h $(ConfigFiles)

HelpFiles = $(MostHelpFiles) $(INCL)/var.h $(INCL)/debug.h

SearchBinFiles = algorithm.o include.o interface.o kernel.o open.o primitive.o

ProbmBinFiles = problem.o

#
# main program
#
all: $(HelpFiles) $(SearchBinFiles) $(ProbmBinFiles)
        $(CC) $(CFlags) $(SearchBinFiles) $(ProbmBinFiles) $(LoadFlags) -o $(CSP)

cleanall:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c

#
# algorithm/
#
algorithm.o: $(HelpFiles) $(ALG)/*.c
        cat $(HelpFiles) $(ALG)/*.c > _;\
        mv -f _ .algorithm.c;\
        $(CC) $(CFlags) -c .algorithm.c -o algorithm.o

#
# include/
#
include.o: $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c
        cat $(MostHelpFiles) $(INCL)/var.c $(INCL)/debug.c > _;\
        mv -f _ .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

#
# interface/
#
interface.o: $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c
        cat $(PROBM)/output.h $(HelpFiles) $(INTFC)/*.c > _;\
        mv -f _ .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

#
# kernel/
#
kernel.o: $(HelpFiles) $(KERNEL)/*.c
        cat $(HelpFiles) $(KERNEL)/*.c > _;\
        mv -f _ .kernel.c;\
        $(CC) $(CFlags) -c .kernel.c -o kernel.o

#
# open/
#
open.o: $(HelpFiles) $(OPEN)/*.c
        cat $(HelpFiles) $(OPEN)/*.c > _;\
        mv -f _ .open.c;\
        $(CC) $(CFlags) -c .open.c -o open.o

#
# primitive/
#
primitive.o: $(HelpFiles) $(PRIM)/*.c
        cat $(HelpFiles) $(PRIM)/*.c > _;\
        mv -f _ .primitive.c;\
        $(CC) $(CFlags) -c .primitive.c -o primitive.o

#
# Problem Depedent Part
#
problem.o: $(HelpFiles) $(PROBM)/*.c
        cat $(HelpFiles) $(PROBM)/*.c > _;\
        mv -f _ .problem.c;\
        $(CC) $(CFlags) -c .problem.c -o problem.o
```